L'Approche du portfolio d'algorithmes pour la construction des algorithmes robustes et adaptatifs

Yanik Ngoko

UNIVERSITE DE GRENOBLE
UNIVERSITE DE YAOUNDE I

THESE

pour obtenir le grade de

DOCTEUR DE L'Université de Grenoble

***Spécialité :* " Informatique"**

préparée dans le cadre d'une cotutelle entre **l'Université de Grenoble**

et

l'Université de Yaoundé I

présentée et soutenue publiquement

par

Yanik Ngoko

le 27 Juillet 2010

L'Approche du portfolio d'algorithmes pour la construction des algorithmes robustes et adaptatifs

---

Directeur de thèse : Denis Trystram

Co-Directeur de thèse : Emmanuel Kamgnia

JURY

ERIC GAUSSIER	Université Joseph Fourier	Président
BERNARD PHILIPPE	INRIA Rennes Bretagne Atlantique	Rapporteur
ALAIN BUI	Université de Versailles Saint-Quentin	Rapporteur
ALFREDO GOLDMAN	Université de São Paulo	Examinateur
DENIS TRYSTRAM	Grenoble INP	Examinateur
EMMANUEL KAMGNIA	Université de Yaoundé	Examinateur

Thèse préparée au sein du Laboratoire d'Informatique de Grenoble avec l'Ecole Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique et au sein du département d'Informatique de l'Université de Yaoundé I

A mes parents : Noukoupoue Christophe et Noukiachon Celestine

Remerciements

Le soutien et la présence de nombreuses personnes ont déterminé l'aboutissement de ce travail de thèse.

Tout d'abord, merci à toi Denis, pour m'avoir proposé ce travail de thèse. Plus d'une fois j'ai voulu reculer tant l'ampleur de la tâche me semblait énorme. Mais tes excellents jugements scientifiques et qualités humaines m'ont permis de tenir la barre.
Merci à toi Emmanuel pour avoir solidement cru en mes capacités à faire de la recherche. C'est cette confiance qui a établi dès mon Master le cheminement que j'ai suivi après.
Merci à toi Alfredo, "le troisième directeur", pour m'avoir sorti de l'isolement dans une période assez trouble. C'est grâce à toi que les idées de base de cette thèse ont pu être formulées. Tu m'as aussi appris à concilier mon travail de recherche et la vie qui coule. Le *Vercors* s'en souvient.
Merci à vous Alain, Bernard et Eric pour avoir accepté de participer à ce jury de thèse. En particulier Bernard, je te remercie en plus pour l'investissement dont tu as fais preuve pour la recherche à l'université de Yaoundé. J'espère qu'à travers cette thèse tu obtiendras une satisfaction.
Merci à Marin et à P-F qui sont sans aucun doute deux grands théoriciens à compter dans cette thèse. Sans vous et la *guess approximation*, je pense que l'histoire de ce document aurait été toute autre.
Merci à vous Paulin (le bosseur cool), Donatien (Madame tout simplement), Innocent (le gentil révolté), Hamza (le gestionnaire du temps), Norbert (le rieur), Meyems (la grande dame), Germaine (l'esprit), Serge (l'épicurien), Etienne (le gentil trop sérieux), Blaiso (le fûté), mes fidèles compagnons de thèse de l'Université de Yaoundé I et du *demi-demi*. Je pense que je vous dois encore le *demi-demi* de soutenance de thèse.
Merci à Fanny, Erik pour m'avoir guidé et soutenu pour la clarification de ma problématique de thèse.
Merci à Lucas (le geeks), Christophe (le duc), Jerôme (le gentil), Jerôme (le pragmatique), Augustin (le comte), Adrien (le Calife philosophe), Daniel (le détendu souriant) mes collègues de bureau de Montbonnot dont les performances n'ont pas été des moindres : "Supporter mon bavardage excessif et mon mutisme subit !"
Merci à Slim (le penseur du possible), Rodrigue (le nihiliste), Pierre (le disponible), Leonardo (la vie est belle), Alfonso (le thé aux sourires), Daouda (la bonne humeur), Marc (tu nous donneras le second *principle* ?), Ahmed (le sourire et l'écoute), Xavier (le cool), Christian (le brun ténébreux), Annie-Claude (la compréhensive), Jean Noel (le café et le work stealing), Swann (gentil sans en avoir l'air) pour avoir accepté que je m'introduise incessamment dans vos bureaux pour discourir des prochaines révolutions.
Merci à J-F et à Jean-Marc pour votre *zen attitude* et votre disponibilité à m'aider notamment dans la présentation de mes idées et les choix scientifiques.
Merci à Iulia, Min, Elvis mes amis de la houille, ainsi qu'a Gustavo, Helmut, Michel, Wellington, Magged mes amis de l'entrée du portoria 3. Grâce à vous, je sais entre autre que le fromage est bon (mais le vert ...), les *Kebabs* aussi, que les montagnes, la mer, les lacs et océans émerveilent, qu'il est bon de chanter à plusieurs, qu'il ne sert à rien de chercher ce qui nous tue, que la *Cachaca* est excellente, qu'on peut s'amuser énormément en travaillant et que je peux compter sur des gens à travers le monde.

Merci à la famille du LIG, du département d'Informatique de l'UYI et du laboratoire de calcul distribué de l'USP.
Merci enfin à ma famille qui m'a toujours poussée vers l'avant. Oui père, tu peux signer que tu l'as enfin ton docteur.

Résumé

Sur plusieurs problèmes il est difficile d'avoir un seul algorithme qui résout optimalement (en temps d'exécution) toutes ses instances. Ce constat motive l'élaboration des approches permettant de combiner plusieurs algorithmes résolvant le même problème. Les approches permettant la combinaison d'algorithmes peuvent être mise en oeuvre au niveau système (en construisant des bibliothèques, des langages et composants adaptatifs etc.) ou au niveau purement algorithmique.
Ce travail se focalise sur les approches génériques de combinaison d'algorithmes au niveau algorithmique avec en particulier l'approche du portfolio d'algorithmes. Un portfolio d'algorithmes définit une exécution concurrente de plusieurs algorithmes résolvant un même problème. Dans une telle exécution, les algorithmes sont entrelacées dans le temps et/ou l'espace. Sur une instance à résoudre, l'exécution est interrompue dès qu'un des algorithmes trouve une solution.

Nous proposons dans cette thèse une classification des techniques de combinaison d'algorithmes. Dans celle ci nous précisons pour chaque technique le contexte le plus adapaté pour son utilisation. Nous proposons ensuite deux techniques de construction des portfolio d'algorithmes.
La première technique est basée sur une adaptation de la méthode des plus proches voisins en apprentissage automatique pour la combinaison des algorithmes. Cette technique est adaptative car elle essaie sur chaque instance de trouver un sous ensemble d'algorithmes adaptés pour sa résolution. Nous l'appliquons dans la combinaison des algorithmes itératifs pour la résolution des systèmes linéaires et nous montrons sur un jeu d'environ mille matrices creuses qu'elle permet de réduire le nombre d'itérations et le temps necessaire dans la résolution. En outre, sur certains jeux d'expérimentations, ces résultats montrent que la technique proposée peut dans la plupart des cas trouver l'algorithme le plus adapté à sa résolution.
La seconde technique est basée sur le problème de partage de ressouurces que nous formulons. Etant donnés, un problème cible, un jeu de données le représentant, un ensemble d'algorithmes candidats le résolvant et le comportement en temps d'exécution du jeu de données sur les algorithmes candidats, le problème de partage de ressources a pour objectif de trouver la meilleure répartition statique des ressources aux algorithmes candidats de sorte à minimiser en moyenne le temps de résolution du jeu de données cibles. Ce problème vise à trouver une solution en moyenne plus robuste que chacun des algorithmes candidats pris séparémment. Nous montrons que ce problème est NP-complet et proposons deux familles d'algorithmes approchés et exacts pour le résoudre. Nous validons les solutions proposées en prenant des données issues d'une base de données pour SAT. Les résultats obtenus montrent que les solutions proposées permettent effectivement de bénéficier de la complémentarité des algorithmes résolvant un même problème pour la construction des algorithmes robustes.

Abstract

On many problems, it is hard to find an algorithm that solves all its instances with the shortest execution time. We are interested here by the design of approaches for combining several algorithms solving the same problems in order to determine a good combination over the whole set of instances. Such approaches can be developed at a system level (with libraries, adaptive languages and components, etc.) or at an algorithmic level.
In this work we are interested on generic algorithmic approaches for combining algorithms. We mainly focus on approaches based on the algorithm portfolio concept. An algorithm portfolio defines a concurrent execution of many algorithms solving a same problem. The execution of algorithms is interleaved in time or in space. The execution is interrupted on the instance to solve when one algorithm finds a solution.

We propose in this thesis a classification of existing techniques for combining algorithms. In this classification, we define for each technique the most suitable context for its utilisation. Then, we propose two techniques for designing porftolio of algorithms.
The first technique is based on an adaptation of the nearest neighbour method in machine learning for the combination of algorithms. This technique is adaptive because for each instance to solve, we aim at determining the most suitable subset of algorithms that can perform well on it. We apply this technique to the resolution of sparse linear systems of equations with iterative solvers and we show experimentally on around one thousand matrices that this technique can effectively be used to reduce the number of iterations and the execution time necessary on the resolution. Moreover, on some experimentations, we were able to predict on almost all the cases the well suited algorithms for solving instances.
The second technique is based on a formulation of the resource sharing problem. Given a target problem, a benchmark of problem instances, a set of candidate algorithms, and the behaviour in execution time of candidate algorithms on the benchmark, the objective in the resource sharing is to find the best way of deploying ressources to candidate algorithms in order to minimize of the execution time necessary to benchmark instances. Our objective with this problem is to build a robust algorithm in sharing ressources to candidate algorithms that are executed in the portfolio. We show that the formulated problem is NP-complete and we propose two families of heuristics and exact algorithms to solve it. Then, we validate our algorithms on a database of SAT problems. The obtained results show that the proposed solutions can serve effectively to exploit complementarities between algorithms in order to propose more robust algorithms.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte et motivations

L'évolution de l'algorithmique et des architectures d'ordinateurs est observable de part la prolifération des bibliothèques spécialisées dans la résolution de différents problèmes et prenant en compte diverses technologies d'ordinateurs. Un constat que l'on peut en tirer est celui du nombre croissant d'algorithmes résolvant un même problème cible. Ceci est par exemple le cas sur les problèmes de tri [Bida and Toledo 2006, Li et al. 2004], de multiplication matricielle [Bilmes et al. 1997, Whaley et al. 2001, Goto and van de Geijn 2008]
[Nasri and Trystram 2004, Li et al. 1997, Hunold et al. 2004], de la résolution des systèmes linéaires et non linéaires [Ern et al. 1994, Saad 2003, Houstis et al. 2000, Barrett et al. 1994, Kelley 1995] ou des problèmes d'ordonnancement [Leung et al. 2004].
Le nombre important d'algorithmes disponibles pour la résolution d'un même problème pose aujourd'hui la question de leur exploitation efficace. Sur diverses architectures, les performances des algorithmes peuvent varier énormément à travers les instances du problème cible [Ern et al. 1994, Hunold et al. 2004]. Pour un utilisateur, non expert du problème cible il se pose alors la question du choix de l'algorithme le mieux adapté à son usage suivant ses cibles de performance.

Une approche qui a souvent été privilégiée dans le choix du "bon" algorithme est celle de la meilleure complexité asymptotique dans le pire ou en moyenne. A cet effet, on peut employer les techniques d'analyse purement théoriques [Knuth 1998, Cormen et al. 2001] ou prenant en compte les évaluations expérimentales [McGeoch et al. 2000]. Se référer à la complexité asymptotique comporte néanmoins des limites. L'analyse dans le cas moyen est sur beaucoup d'algorithmes assez difficile à établir ; Elle demeure en outre très probabiliste sur la distribution des données en entrée qui n'est pas toujours facile à connaître. L'analyse du pire des cas quant à elle peut donner une borne trop grossière sur l'efficacité réelle des différents algorithmes car en général, elle ne prend en compte que certaines instances du problème.
Une autre approche pour faire face à ce problème a été celle du développement par problèmes, d'une réponse propre permettant de combiner au mieux différents algorithmes résolvant le

même problème, notamment avec les concepts d'algorithmes hybrides et poly-algorithmes [Cung et al. 2006, Frigo and Johnson 1998, Nasri and Trystram 2004, Ngoko 2006]. L'importance des problèmes pour lesquels il existe différents algorithmes employables nécessite toutefois que ces techniques soient rendues de nos jours à un niveau automatisable. L'exigence d'automatisation signifie en particulier que les solutions proposées indiquent comment dériver automatiquement une solution de combinaison d'algorithmes sur une large classe d'algorithmes.

1.2 Positionnement et objectifs de la thèse

De nombreux travaux se sont intéressés à la combinaison automatique des algorithmes. Les solutions qui y ont été proposées peuvent être situées à deux principaux niveaux :

- **Les solutions au niveau algorithmique** proposant une définition du problème informatique de la combinaison automatique. Dans cette logique, ont été introduit : le problème de sélection des algorithmes [Rice 1979], des problèmes de cascading d'algorithmes [Frigo and Johnson 1998, Nasri and Trystram 2004, Bida and Toledo 2006] ainsi que d'autres modélisations du problème de combinaison reposant sur des portfolio d'algorithmes [Sayag et al. 2006, Streeter et al. 2007]. Dans les solutions proposées ici, la combinaison automatique des algorithmes est perçue comme étant un problème particulier auquel il faut proposer un algorithme de résolution (un poly-algorithme).
- **Les solutions au niveau système** proposant une architecture système pour la combinaison des algorithmes. Ici, on peut distinguer : le système LSA [Gannon et al. 2000] proposant une infrastructure permettant à l'utilisateur de composer différents algorithmes, l'approche SANS [Dongarra et al. 2006b, Demmel et al. 2006]
  [Blackford et al. 1997, Berman et al. 2001, Chen et al. 2003] proposant une architecture de composants permettant d'utiliser plusieurs algorithmes dans la résolution d'un problème ou les approches dynamiques [Roch et al. 2006, Buisson et al. 2005] proposant une combinaison des algorithmes dépendante des évènements qui surviennent sur la plate-forme d'exécution.

Dans cette thèse, nous nous intéressons à des solutions au niveau algorithmique pour combiner plusieurs algorithmes résolvant un même problème afin de minimiser le temps de résolution. Notre objectif est de proposer, étant donné un problème solvable par une liste connue d'algorithmes, des modèles permettant de dériver des algorithmes plus robustes et adaptatifs en se basant sur ceux connus.
Cet objectif peut être décrit plus formellement comme suit : Considérons les instances $\mathcal{I}$ d'un problème et la liste $\{A_1, \ldots, A_k\}$ des algorithmes le résolvant. Soit $Algo$, l'algorithme que nous espérons dériver de $\{A_1, \ldots, A_k\}$ et $C(A_i, I_j)$ le temps de résolution d'une instance I_j avec l'algorithme A_i. L'objectif de robustesse indique qu'en moyenne, le temps d'exécution de notre algorithme dérivé soit meilleur que celui de n'importe quel algorithme pris séparément. Soit alors : $\sum_{I_j \in \mathcal{I}} C(Algo, I_j) \leq \sum_{I_j \in \mathcal{I}} C(A_i, I_j)$. L'objectif d'adaptativité veut que l'algorithme dérivé s'adapte à chaque instance qu'il résout de sorte à se rapprocher de la meilleure performance algorithmique possible. Ceci indique que pour chaque instance I_j la valeur $C(Algo, I_j)$

soit minimale. Ce dernier objectif englobe la robustesse mais est plus difficile à atteindre en général.

1.3 Contributions et guide de lecture

Dans cette thèse nous nous intéressons à la modélisation algorithmique des techniques de combinaison d'algorithmes. La thèse principale que nous défendons est celle de promouvoir l'usage des approches par portfolio d'algorithmes dans la combinaison automatique des algorithmes. Les portfolio d'algorithmes ont été introduits dans [Huberman et al. 1997] et s'inspirent du problème on-line de sélection des portfolio [Borodin and El-Yaniv 1998]. Un portfolio d'algorithmes définit une exécution concurrente de plusieurs algorithmes résolvant un même problème. Dans une telle exécution, les algorithmes sont entrelacées dans le temps et/ou l'espace. Sur une instance à résoudre, l'exécution est interrompue dès qu'un des algorithmes trouve une solution.
Nous considérons en particulier dans cette thèse, les modèles de portfolio d'algorithmes par partage de ressources et de temps introduit dans [Sayag et al. 2006] et auxquels nous ajoutons des considérations pratiques. L'approche générale de résolution d'une instance avec les portfolio d'algorithmes ici comporte les phases suivantes :

- La sélection du sous ensemble d'algorithmes $\mathcal{A}^I$ qui a priori sur l'instance I donnera le plus petit temps d'exécution possible
- La détermination du meilleur portfolio d'algorithmes entrelaçant les exécutions des algorithmes dans $\mathcal{A}^I$
- L'exécution de ce portfolio afin de résoudre I

Nous démontrons l'efficacité de cette méthodologie en prenant deux approches permettant de construire des algorithmes hybrides et adaptatifs basées sur les portfolio d'algorithmes.

Ce manuscrit est organisé en 6 chapitres. Le chapitre 2 présente une vue d'ensemble des techniques de combinaison d'algorithmes en insistant en particulier sur les portfolio d'algorithmes. Il introduit une classification des techniques existantes. Deux classes principales sont ici considérées en fonction de l'hypothèse selon laquelle l'on puisse avoir un modèle analytique de prédiction des performances des algorithmes que l'on veut combiner.

Le chapitre 3 propose une approche par portfolio d'algorithmes s'appuyant sur les techniques d'apprentissage automatique et en particulier la méthode des plus proches
voisins [Mitchell 1997]. La différence essentielle entre le modèle qui y est présenté et ceux existants dans la littérature est dans la possibilité d'exécution concurrente des algorithmes. L'approche proposée ici est validée sur le problème de résolution des systèmes linéaires en prenant un ensemble de 959 matrices issus d'une collection de matrices creuses de l'université de Floride [1].

Le chapitre 4 propose une approche par portfolio d'algorithmes dans un contexte parallèle et homogène. Cette approche est inspirée du problème de partage continu de ressources pro-

[1] `http://www.cise.ufl.edu/research/sparse/matrices/`

posés dans [Sayag et al. 2006]. Afin de le rendre plus réaliste, nous le modifions en considérant que les ressources sont discrètes. Nous présentons des résultats de complexité sur le problème de partage discret des ressources ainsi que des algorithmes exacts et approchés. Nous évaluons les algorithmes et l'intérêt du problème de partage de ressources en effectuant des simulations avec une base de données(SatEx) contenant les performances de solveurs sur le problème SAT [Simon and Chatalic 2001].

Le chapitre 5 traite du modèle de partage discret de ressources avec l'hypothèse que tous les algorithmes doivent nécessairement participer au partage. Nous proposons sur ce modèle des résultats de complexité et des algorithmes de résolution exacts et approchés. Nous validons ensuite les algorithmes proposés en effectuant des simulations sur la base de données SatEx.

Enfin, le chapitre 6 propose une évaluation générale des deux approches par portfolio d'algorithmes que nous avons proposées. Nous présentons sommairement les résultats obtenus, analysons l'adéquation de ces approches en fonction du contexte générale de la combinaison des algorithmes et relevons les faiblesses des différentes approches. Nous terminons en présentant les principales perspectives que nous envisageons suite à ce travail.

Chapitre 2

Modélisation

Résumé : *Dans ce chapitre nous présentons différentes modélisations visant à combiner les algorithmes résolvant un problème avec pour objectif de minimiser le temps d'exécution. Nous proposons en particulier une classification des techniques existantes en insistant en particulier sur les approches par portfolio d'algorithmes.*

2.1 Introduction

Ce chapitre est consacré à la présentation des techniques de combinaison automatique d'algorithmes résolvant un même problème. Les approches de combinaison d'algorithmes sont variées et peuvent être situées au niveau algorithmique et au niveau système. Dans tous les cas, le but de la combinaison est de produire sur chaque instance à résoudre un ordre d'exécution des algorithmes dans le temps et dans l'espace. Au niveau algorithmique on peut construire cet ordre comme étant la solution à un problème informatique posé. Nous verrons quelques uns de ces problèmes dans la suite. Au niveau système, on joint à l'ordre d'exécution un environnement d'exécution contenant par exemple un ordonnanceur de tâches afin de contrôler l'exécution de l'instance. Ceci est intéressant car plusieurs évènements peuvent pertuber les estimations initiales faites sur la qualité de l'ordre d'exécution défini initialement pour une instance. Il est important dans ces cas de pouvoir adapter l'exécution de l'instance à ceux ci. Nous ciblons uniquement les approches au niveau algorithmique visant à réduire le temps d'exécution en moyenne et par instances du problème cible.

Plusieurs vues d'ensemble de la littérature sur la combinaison des algorithmes ont été proposées. Dans [Cung et al. 2006], les auteurs proposent une classification des algorithmes hybrides et adaptatifs ainsi qu'une technique originale de couplage d'algorithmes dans un contexte parallèle et dynamique. Une autre classification des algorithmes hybrides et adaptatifs a été proposé dans [Gagliolo and Schmidhuber 2006]. Celle ci en particulier distingue les moments et les différents lieux où peuvent prendre place le couplage d'algorithmes. Dans les deux études que nous avons relevées, la classification proposée couvre peu le champ des techniques de

mise en oeuvre permettant de combiner les algorithmes. Dans [Dongarra et al. 2006b], les auteurs proposent un survol des techniques de sélection et de calibrage d'algorithmes pour la résolution des problèmes numériques. Ce survol qui inclut la mise en oeuvre au sein d'un système des approches de combinaison d'algorithmes couvre très peu le niveau algorithmique. Dans [An et al. 2003, Guo 2003], d'autres vues d'ensemble des techniques de combinaison d'algorithmes sont présentées. Seulement, celles ci restent très focalisées à des méthodologies précises de combinaison d'algorithmes basées sur l'apprentissage automatique.

Dans ce chapitre, nous proposons une nouvelle classification des approches de combinaison d'algorithmes. A travers celle ci, nous donnons en comparaison avec les autres travaux existants, une vue plus centrée sur le niveau algorithmique et couvrant un plus large spectre d'approches. La classification que nous proposons distingue les approches de combinaison d'algorithmes en approches clairvoyantes et non-clairvoyantes en fonction de la connaissance ou non a priori d'une estimation du temps d'exécution. Pour chaque classe d'approches, nous analysons l'adaptation des hypothèses sous jascentes pour la combinaison des algorithmes numériques et combinatoires et nous présentons quelques exemples d'applications. La différence principale entre les classes clairvoyantes et non clairvoyantes est dans l'hypothèse selon laquelle on peut avoir une métrique de performance analytique donnant le comportement des différents algorithmes à combiner sur les instances du problème cible. Nous nous focalisons plus sur les approches non-clairvoyantes où nous montrons l'intérêt d'utiliser les approches par portfolio d'algorithmes dans ce cas.
L'idée principale dans les portfolio d'algorithmes est d'autoriser l'exécution concurrente de plusieurs algorithmes résolvant un même problème. Si cette approche à première vue souffre de la redondance des calculs nécessaires à la résolution d'un problème, nous montrons que cette redondance peut être exploitée comme un temps consacré en réalité à la sélection du meilleur algorithme.

La suite du chapitre est organisée comme suit : Dans la section 2.2, nous présentons les approches clairvoyantes de combinaison automatique d'algorithmes. La section 2.3 présente les approches non clairvoyantes avec en particulier les approches par portfolio d'algorithmes. Nous tirons un bilan général et présentons les perspectives dans la section 2.4.

2.2 Approches clairvoyantes de combinaison d'algorithmes

Dans cette section, nous présentons les approches clairvoyantes de combinaison d'algorithmes. Nous illustrons ces approches en prenant exemple sur des algorithmes résolvant les problèmes d'algèbre linéaire dense et les problèmes combinatoires polynomiaux. Nous précisons que ces domaines ont suscité beaucoup d'intérêts dans la combinaison des algorithmes notamment sur la multiplication matricielle et le tri [Whaley et al. 2001, Bilmes et al. 1997]
[Houstis et al. 2000, Goto and van de Geijn 2008, Bida and Toledo 2006].

2.2.1 Préliminaires

Les approches clairvoyantes sont basées sur la notion de poly-algorithme. Un poly-algorithme est un algorithme composé de plusieurs algorithmes résolvant le même problème. Lors de la résolution d'une instance du problème, il choisit d'exécuter un des algorithmes qui le compose. Les approches clairvoyantes sont basées sur le scénario 2.2.1 décrit comme suit :

Scénario 2.2.1. *Un utilisateur est face à un problème donné par une infinité d'instances et un nombre fini d'algorithmes (en général faible sur le problème cible). L'objectif est de construire un poly-algorithme pour cet utilisateur qui sur chaque instance du problème va combiner ou sélectionner les différents algorithmes de sorte à résoudre les instances du problème avec le plus petit temps d'exécution possible. Plus formellement, étant donné un problème calculable P et un ensemble d'algorithmes $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$, on veut construire un poly-algorithme $Poly(.)$ basé sur $\mathcal{A}$ tel que quelle que soit instance I de P, $Poly(I)$ la résolve avec le plus petit temps d'exécution possible.*

Le Scénario 2.2.1 a été classiquement utilisé dans le calcul haute performance pour construire des algorithmes adaptatifs [Dongarra et al. 2006b, Whaley et al. 2001, Goto and van de Geijn 2008] [Nasri and Trystram 2004, Frigo and Johnson 1998, Bida and Toledo 2006, Li et al. 1997] [Ngoko 2006].

Les approches clairvoyantes admettent l'existence d'un modèle d'estimation du temps d'exécution des algorithmes sur les instances. Cette hypothèse est raisonnable sur plusieurs problèmes notamment ceux de l'algèbre linéaire dense et les problèmes combinatoires polynomiaux. En effet, la littérature autour des noyaux fondamentaux dans ces domaines (multiplication matricielle, tri) est particulièrement riche en algorithmes pour lesquels les modèles analytiques de la performance ont été développés [Li et al. 1997, Ngoko 2006, Knuth 1998]. La performance analytique des algorithmes est souvent ici donnée en fonction de la taille de l'instance du problème à résoudre et des paramètres architecturaux de l'environnement d'exécution. Sur le produit de matrices, par exemple, la taille du cache L1, le nombre de registres ou la forme(triangulaire, carrée) de la matrice sont des paramètres architecturaux souvent considérés. Dans plusieurs cas, la précision des modèles analytiques proposés a été validée à travers des études expérimentales [Knuth 1998, Whaley et al. 2001, Bilmes et al. 1997, Li et al. 1997].

L'existence d'un modèle d'estimation du temps d'exécution des instances sur les algorithmes est fondamentale dans les approches clairvoyantes. Dans la suite, nous allons présenter deux types d'approches basées sur cette hypothèse et suivant le scénario ci dessus. Ce sont les approches par sélection d'algorithmes et celles par combinaison récursive d'algorithmes.

2.2.2 La sélection d'algorithmes

Les approches par sélection d'algorithmes sont inspirées des modèles de sélection d'algorithmes introduit par John Rice [Rice 1979]. Ici, on construit $Poly(.)$ en créant une fonction de

placement qui doit être capable d'associer à chaque instance I de P un algorithme dans $\mathcal{A}$. Une exigence pour avoir une telle fonction est la capacité de prédire pour chaque instance son temps d'exécution sur les algorithmes. En supposant que $Perf(A_j, I)$ donne le temps d'exécution de l'algorithme A_j sur l'instance I, la fonction de placement sur l'instance I va exécuter l'algorithme A_{opt} qui minimise $Perf(A_j, I)$.

La sélection d'algorithmes est une approche naturelle pour la combinaison d'algorithmes. Elle est simple et dépend principalement de la précision de la fonction $Perf$. Un avantage d'utilisation de cette approche en algèbre linéaire dense ou sur les problèmes polynomiaux est que dans ces cas, les principaux paramètres souvent utilisés dans la construction de la fonction $Perf$ sont ici la taille de l'instance à résoudre et certains paramètres machines que l'on peut déterminer à l'avance. Dans ces cas donc, on peut décider du meilleur algorithme pour la fonction $Perf$ sur chaque instance en un temps très court. Ainsi si $time(Poly(I))$ est le temps d'exécution de $Poly(.)$ sur I et $time(A_{opt}(I))$ est le temps d'exécution optimal d'un algorithme dans $\mathcal{A}$ (au sens de $Perf$) sur l'instance I, on a $time(Poly(I)) = time(A_{opt}(I)) + C(I)$ où $C(I)$ est le temps necessaire à l'évaluation de la fonction de performance sur I.

Dans la suite, nous montrons comment la sélection des algorithmes peut être appliquée pour automatiser la combinaison des algorithmes de multiplication parallèle de matrices.

2.2.2.1 Exemple : Sélection d'algorithmes pour la multiplication parallèle des matrices

Nous nous référons ici à des études effectuées dans [Li et al. 1997, Ngoko 2006]. Il existe aussi des études similaires que l'on peut retrouver dans [Nasri and Trystram 2004]
[Hunold et al. 2004, Blackford et al. 1997]. Considérons le calcul de $C = A \times B$ où $A \in \mathbb{R}^{m \times k}$ et $B \in \mathbb{R}^{k \times n}$ dans un contexte parallèle et distribué. Nous utiliserons ici trois algorithmes. Le premier est une adaptation de l'algorithme systolique en dimension 2 (où la matrice C est stationnaire) sur une grille carrée de processeurs ; Il a été proposé dans [Li et al. 1997]. Le second algorithme a été proposé dans [Agarwal et al. 1994] et est basé sur une formulation bloc du produit externe des matrices. Le troisième algorithme a été proposé dans [Grayson and van de Geijn 1996] et est basé sur une parallélisation bloc de l'algorithme de Strassen. L'algorithme de Strassen peut être exécuté avec plusieurs niveaux de récursivité. Nous considérons ici seulement un niveau de récursivité.
Une analyse détaillée de ces différents algorithmes a été proposée dans [Li et al. 1997]
[Ngoko 2006]. Cette analyse propose des fonctions analytiques de coût des différents algorithmes en supposant que les communications répondent au modèle LogP [Culler et al. 1996] efficace pour caractériser les coûts de communication en environnement distribué. Désignons par $t_{add}(x), t_{mult}(x)$ et $t_{comm}(x)$ les fonctions de coût analytique d'un de nos algorithmes x. Considérons aussi les notations $\delta, \gamma, \alpha, \beta$ pour désigner respectivement : le coût d'une addition flottante, le coût d'une multiplication flottante, la latence et l'inverse de la bande passante dans le modèle LogP. En supposant que l'on a une grille de $p = r^2$ processeurs, la fonction de coût analytique du premier algorithme est donnée par :

$$t_{add}(1) = \frac{mnk}{p}\delta\ , \quad t_{mult}(1) = \frac{mnk}{p}\gamma,$$

$$t_{comm}(1) = (8 + 2r)\alpha + 2\beta k(\frac{4mk+4kn+r(mk+kn)}{p})$$

Pour les autres algorithmes nous avons :

$$t_{add}(2) = \frac{mnk}{p}\delta\,, \quad t_{mult}(2) = \frac{mnk}{p}\gamma,$$
$$t_{comm}(2) = r(\alpha + \beta\frac{mk}{p})(r-1) + r(\alpha + \beta\frac{kn}{p})(r-1)$$

et

$$t_{add}(3) = (\frac{5mk+5kn+8mn}{4p} + \frac{7mnk}{8p})\delta\,, \quad t_{mult}(3) = \frac{7mnk}{8p}\gamma$$
$$t_{comm}(3) = 7r(r-1)(2\alpha + \frac{\beta(mk+kn)}{4p})$$

On peut noter ici que ces fonctions de coût suggèrent qu'en fonction des matrices et de l'environnement d'exécution, les différents algorithmes n'auront pas la même performance. En effet, le coût des communication est différent pour les trois algorithmes. Par ailleurs, le coût des multiplications et des additions est le même pour le premier et le troisième algorithme. Ces observations ont en outre été vérifiées à travers différentes exécutions des algorithmes ci dessus [Li et al. 1997, Ngoko 2006].
A partir des fonctions de coût ci dessus, on peut construire un poly-algorithme pour la multiplication parallèle des matrices qui en fonction des matrices en entrées et de la plate-forme cible exécutera l'algorithme qui conduit à la minimisation du temps total d'exécution. Un tel poly-aglorithme a été proposé dans [Li et al. 1997] et sa performance est meilleure que celle de chacun des algorithmes pris séparément.

Il existe plusieurs autres exemples en algèbre linéaire dense et sur les problèmes polynomiaux combinatoires similaires à celui que nous avons présenté. Nous renvoyons le lecteur principalement aux études faites dans [Lamarca and Ladner 1999, An et al. 2003, Li et al. 2004, Dongarra et al. 2006b, McCracken et al. 2003, Bilmes et al. 1997, Whaley et al. 2001].

La sélection d'algorithmes est une approche pour automatiser la combinaison de plusieurs algorithmes résolvant le même problème. Elle est simple avec une décision de choix d'algorithmes est peu coûteuse.
L'exemple ci dessus présente toutefois, une combinatoire dans le choix des algorithmes qui n'est pas explorée dans ce cas. En effet, dans l'algorithme de Strassen, la méthode de Strassen n'a été appliquée qu'à un seul niveau de récursivité. Plusieurs autres niveaux d'applications peuvent être considérés avec au bout différentes performances dans la résolution. Cette observation pose la question de la décomposition optimale du problème initial en sous problèmes quand on a un algorithme récursif. Nous allons dans la suite proposer une approche de combinaison d'algorithmes basée sur ce principe.

2.2.3 La combinaison récursive des algorithmes

L'approche de composition récursive que nous proposons a été appliquée principalement pour le calcul de la transformée de Fourier discrète [Frigo and Johnson 1998] et le tri [Bida and Toledo 2006].

Dans la combinaison récursive des algorithmes, nous sommes intéressés par l'usage optimal d'un algorithme diviser-pour-régner à partir de l'observation suivante : étant donné un algorithme récursif A et d'autres algorithmes $\mathcal{A} = \{A_1, \ldots, A_k\}$ résolvant le même problème, on peut substituer tout appel récursif dans A par un appel à un algorithme dans $\mathcal{A}$. Dans la figure 2.1, nous illustrons ce fait. Ici, nous avons un algorithme exécuté à trois niveaux de récursvité sur une instance. Le premier niveau décompose l'instance à résoudre en deux autres. Sur une de ces instances, on applique l'algorithme A_1 et sur la seconde, on applique à nouveau l'algorithme récursif. A la fin des appels récursifs, on doit fusionner les résultats obtenus. Pour cela, nous avons représenté les étapes de pré-processing et post-processing représentant l'exécution de l'algorithme récursif avant et après les appels récursifs.

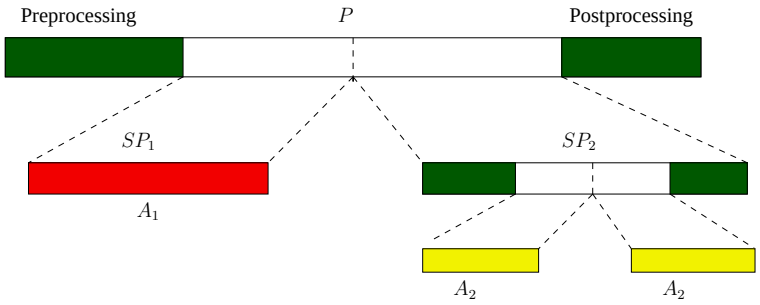


FIG. 2.1 – Combinaison récursive d'algorithmes sur un problème P à résoudre. Celui ci est subdivisé en deux sous problèmes identiques $SP1$ et $SP2$. Le premier sous problème est résolu par l'algorithme A_1. Le second est à nouveau subdivisé en sous problèmes qui sont résolus par A_2.

L'observation ainsi faite indique qu'en fonction de la taille du problème à résoudre et des algorithmes disponibles, il existe plusieurs combinaisons possibles permettant de résoudre le problème cible. L'idée dans la combinaison récursive est donc de trouver pour toute instance à résoudre la décomposition optimale basée sur un ou plusieurs algorithmes récursifs conduisant au plus petite coût de résolution de l'instance. On peut évidemment noter qu'a priori le nombre de telles décompositions peut être très important en fonction de la taille de l'instance.

2.2.3.1 Exemple de combinaison récursive sur le tri

Nous proposons ici un exemple simplifié de la combinaison récursive des algorithmes de tri. Cet exemple est inspiré de l'approche de combinaison récursive développée dans [Lagoudakis and Littman 2000]

Supposons par exemple que nous avons à trier des éléments avec trois algorithmes dans un contexte séquentiel : le tri par insertion (Insertion Sort), le Merge Sort et le Quick Sort. Sur des petites tailles de données $n \leq n_0$ nous pouvons connaître quel est le meilleur algorithme (algorithme le plus rapide) ainsi que son coût d'exécution. Sur des grandes tailles de données on peut exploiter la récursivité des algorithmes de Merge Sort et Quick Sort pour se ramener à un problème sur une taille de données plus petite. Étant donné un tableau de taille n dans ce cas, soit $OPT(n)$ la fonction de coût pour le tri optimal. Nous posons

$$OPT(n) = \min\{OPTM(n), OPTQ(n), OPTI(n)\}$$

Ici, $OPTI(n)$ désigne le coût du tri si on décide d'appliquer le tri par insertion, $OPTM(n)$ donne le coût du tri si on décide d'appliquer le Merge Sort et donc de subdiviser le tableau de taille n en deux sous tableaux de taille $n/2$ et $OPTQ(n)$ le coût du tri si on applique le Quick Sort au premier niveau. Ces deux fonctions par ailleurs peuvent être décrites comme suit :

$$\begin{array}{c} OPTM(n) = M(n) + OPT(\lfloor n/2 \rfloor) + OPT(\lceil n/2 \rceil), \\ OPTQ(n) = Q(n) + \frac{1}{n}\sum_{p=1}^{n-1}(OPT(p) + OPT(n-p)) \end{array}$$

$M(n)$ ici désigne le coût de la fusion du tableau de taille n en deux sous tableaux $M(n) = \theta(n)$. $Q(n)$ désigne le coût du partitionnement d'un tableau de taille n dans l'algorithme de Quicksort. La somme $\sum_{p=1}^{n-1}(OPT(p) + OPT(n-p))$ est liée au fait qu'à la probabilité $\frac{1}{n}$ on peut avoir un découpage du tableau à trier en deux tableaux de taille $p, 1 \leq p \leq n-1$ et $n-p$. Étant donné un tableau de taille n à trier, on peut en utilisant la fonction OPT construire un arbre de décomposition indiquant le meilleur couplage récursif d'algorithmes permettant de le trier. Dans la construction de cet arbre on peut supposer que si on a une décomposition optimale pour une taille n' donnée, celle ci reste optimale si elle est incluse dans une décomposition de taille $n > n'$.

Nous avons présenté quelques approches pour la combinaison automatique des algorithmes en admettant que l'on a un modèle analytique estimant le temps d'exécution des algorithmes sur les instances. L'approche de sélection d'algorithmes est simple et peu coûteuse mais ne prend pas en compte éventuellement certaines compositions d'algorithmes. L'approche de la combinaison récursive considère un espace de recherche beaucoup plus grand mais nécessite un temps d'exécution important pour déterminer la solution optimale. Dans la suite, nous portons un regard critique sur les approches clairvoyantes

2.2.4 Analyse critique des approches clairvoyantes

Nous avons justifié l'hypothèse d'un modèle analytique de la performance des algorithmes sur les approches clairvoyantes en prenant exemple sur les problèmes d'algèbre linéaire dense et les problèmes combinatoires polynomiaux. Considérons maintenant cette hypothèse sur les

problèmes difficiles que sont les problèmes d'algèbre linéaire creuse et des problèmes combinatoires difficiles (principalement les problèmes NP complet comme le problème SAT ou les problèmes d'ordonnancement).

Sur ces problèmes, l'hypothèse de la prédictibilité des temps d'exécution est difficile à vérifier. Ceci est principalement dû au fait qu'il y a plusieurs éléments inconnus ou très coûteux à estimer qui sont déterminants dans la performance des algorithmes sur les instances. Par exemple, le temps d'exécution dans la résolution d'un système linéaire creux avec les méthodes itératives dépend principalement de la distribution des valeurs propres [Saad 2003]. Seulement, le calcul de ces valeurs propres peut être tout aussi coûteux que la résolution du système [Saad 2003]. Sur les heuristiques résolvant les problèmes NP complet par ailleurs, il est très difficile d'estimer le temps d'exécution dans la résolution d'une instance. Quelques pistes intéressantes à cet effet ont été suivies dans [Lobjois and Lemaître 1998] sur le problème de satisfaction des contraintes et dans [Knuth 1975] sur les algorithmes basés sur le *bactracking*. Néanmoins, les techniques utilisées sont très dépendantes des problèmes cibles et comme montré dans[Knuth 1975] [Lobjois and Lemaître 1998], l'estimation peut conduire à une très grande borne supérieure. Enfin, un résultat général sur la difficulté de prédiction des performances des algorithmes peut être trouvé dans [Guo 2003].

Sur tous les types de problèmes, les approches clairvoyantes ne sont pas necessairement adéquates. Dans la suite, nous allons proposer une autre familles d'approches dites non-clairvoyantes qui peuvent servir d'alternatives

2.3 Approches non-clairvoyantes de combinaison

Dans les approches non-clairvoyantes, nous ne supposons pas l'existence d'un modèle d'estimation du temps d'exécution des algorithmes sur les instances. Nous allons présenter dans cette partie trois approches de ce type. Ce sont : les approches on-line, les approches basées sur l'apprentissage automatique et les approches par portfolio d'algorithmes.

2.3.1 Les approches on-line

Les approches dites on-line de combinaison d'algorithmes proposent de combiner les algorithmes sans connaissance a priori de la nature de l'instance à résoudre. Elles ont été principalement étudiées dans [Gagliolo and Schmidhuber 2008, Gagliolo and Schmidhuber 2006, Streeter et al. 2007]. Ici, on procède en général en déclinant le problème de combinaison d'algorithmes en un problème on-line classique [Auer et al. 2002, Borodin and El-Yaniv 1998]. Les problèmes on-line les plus étudiés ici sont le problème du bandit manchot [Auer et al. 2002, Gagliolo and Schmidhuber 2006] et le problème on-line de recherche du plus court chemin [Streeter et al. 2007]. Ces problèmes sont en général composés de trois éléments prin-

cipaux : une séquence d'entrée, un ensemble fini de stratégies à choisir sur chaque entrée, une métrique de coût donnant la récompense occasionnée par le choix d'une stratégie sur une entrée. Ces problèmes sont généralement adaptés à la combinaison des algorithmes en prenant la séquence d'entrée comme étant les instances du problème à résoudre, l'ensemble de stratégies comme étant les différentes combinaisons d'algorithmes applicables et la récompense comme étant le temps d'exécution obtenu par un algorithme sur une instance. Ainsi, une approche de minimisation des récompenses permet de minimiser le temps d'exécution dans la résolution du problème cible.

Les principaux intérêts des approches on-line sont leur générécité et leur simplicité. Toutefois ces approches exploitent très peu d'informations sur le problème cible de sorte qu'il est difficile a priori d'avoir ainsi de bonnes performance en pratique.

2.3.1.1 Sélection on-line des algorithmes par le modèle du bandit manchot

Un modèle de sélection des algorithmes a été proposé dans [Gagliolo and Schmidhuber 2008] à partir d'une adaptation au problème du bandit manchot. Ce problème peut être décrit comme suit :

BANDIT MANCHOT :
Un joueur doit traiter une séquence de n données en employant une action parmi k actions distinctes. A chaque donnée traitée, le joueur peut observer un retour qui peut être un gain ou une perte en fonction de l'action choisie. Si la donnée i est traitée en prenant une action j, $1 \leq j \leq k$, ce gain ou cette perte est noté $C(i, j)$. Si $j_1, \ldots, j_n$ est la séquence des données à traiter, l'objectif du joueur est de maximiser la somme des retours $\sum_{i=1}^{n} C(i, j_i)$. Toutes les instances ici sont traitées une seule fois.

Dans la solution proposée dans [Gagliolo and Schmidhuber 2008], la liste des actions est considérée comme étant les algorithmes et la séquence des données à traiter comme étant les instances du problème pour lequel on veut combiner les algorithmes. Les valeurs $C(i, j)$ sont choisies comme étant l'inverse du temps d'exécution de l'algorithme i sur l'instance j. Avec cette modélisation, le problème de combinaison des algorithmes est résolu avec les approches on-line de résolution du problème du bandit manchot.

Les approches on-line sont une alternative lorsque l'on ne dispose pas d'un modèle donnant le temps d'exécution des algorithmes sur les instances. En pratique toutefois, les solutions qui y sont proposées sont probabilistes et dépendent fortement de la distribution d'arrivée des instances qui est difficile à connaître. Dans la suite, nous allons proposer deux approches qui prennent plus en compte les spécificités liées au problème. Dans celles ci, le scénario de combinaison des algorithmes est légèrement modifié. Nous commençons par le présenter.

2.3.1.2 Autre Scénario pour la combinaison des algorithmes

Sans modèle de prédiction des performances algorithmique, il est difficile de développer une base pour définir la combinaison des algorithmes résolvant le même problème. Pour avoir une idée de la performance des algorithmes, une approche consiste à l'observer dans un premier temps sur un sous ensemble fini d'instances puis à extrapoler l'observation faite à l'ensemble des instances du problème. Avec cette considération, la combinaison des algorithmes peut être pensée dans le scénario 2.3.1 décrit comme suit :

Scénario 2.3.1. *Étant donné un ensemble fini d'algorithmes $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$, nous avons un utilisateur face à un problème $\mathcal{P}$ à résoudre qui comporte un ensemble fini connus d'instances $\mathcal{I} = \{I_1, \ldots, I_n\}$. L'objectif est de construire un poly-algorithme $Poly(.)$ pour cet utilisateur en exploitant les connaissances sur $\mathcal{P}$ et $\mathcal{I}$ de sorte à combiner efficacement les algorithmes dans $\mathcal{A}$ de sorte à minimiser le temps d'exécution obtenu dans la résolution des instances.*

Dans le scénario 2.3.1, un ensemble fini d'instances est introduit afin d'avoir une idée de la performance des algorithmes sur les instances. Cette hypothèse peut être justifiée par deux autres éléments en pratique. Ce sont :

– La notion de benchmark dans la résolution des problèmes
– La nature des instances régulièrement résolues par un utilisateur.

Étant donné un problème calculable, un benchmark est un ensemble fini d'instances qui sont supposées être représentatives de la difficulté de résoudre le problème. Quelques exemples fameux de benchmark sont le benchmark LINPACK [Dongarra et al. 2003] et le benchmark NAS [Kamath and Weeratunga 1990] qui sont utilisés pour évaluer la performance des architectures d'ordinateurs. Les benchmarks sont de nos jours régulièrement utilisés pour évaluer la performance des algorithmes. Sur les heuristiques résolvant les problèmes NP complet, cette tendance est plus accentuée comme le montre le développement massif des benchmarks dans ce domaine avec les librairies OR [1], CSPLIB [2] ou GGEN [3]. En prenant donc l'ensemble $\mathcal{I}$ comme étant un benchmark du problème à résoudre, on peut espérer capturer à travers celui ci la performance en moyenne des algorithmes sur les instances du problème. Si le choix d'un tel exemple peut sans doute toujours être discuté en pratique, nous notons qu'il peut tout le temps être amélioré en ajoutant ou en retirant des instances de $\mathcal{I}$. Une autre motivation pour considérer un ensemble fini d'instances est que celui ci pourrait être adapté aux instances régulièrement résolues par un utilisateur. En pratique en effet, l'activité d'un utilisateur pour lequel on veut construire un mécanisme de combinaison d'algorithmes ne prend pas nécessairement en compte l'ensemble complet des instances du problème. Dans la résolution des systèmes linéaires par exemple, un utilisateur travaillant dans le domaine de l'astrophysique ne manipulera pas au quotidien les mêmes matrices qu'un utilisateur travaillant dans l'ingénierie chimique. Il est raisonnable dans ces cas de penser à extraire un sous ensemble d'instances que l'on considère. Le Scénario 2.3.1 diffère du premier (scénario 2.2.1) par l'introduction d'un ensemble fini d'ins-

[1] http ://people.brunel.ac.uk/ mastjjb/jeb/info.html
[2] http ://www.csplib.org/
[3] http ://ggen.ligforge.imag.fr/

tances pour saisir le comportement des algorithmes. Dans la suite, nous allons présenter deux approches basées sur ce scénario.

2.3.2 Approches basées sur l'apprentissage

Les approches basées sur l'apprentissage [Dongarra et al. 2006b, Bhowmick et al. 2006] [Xu et al. 2008, Houstis et al. 2000, Guo 2003, Li et al. 2004] sont inspirés des techniques d'apprentissage automatique [Mitchell 1997]. La plupart d'entre elles s'appuient sur un modèle de sélection des algorithmes proposés par John Rice [Rice 1979] il y a longtemps. Dans ce modèle, chaque problème à résoudre est caractérisé par un certains nombres d'attributs. Les attributs sont censés avoir un impact sur sa performance avec les algorithmes. Dans la résolution d'un système linéaire par exemple, des attributs peuvent être le nombre de valeurs non nulles, la forme de la matrice (*ratio aspect*), son conditionnement, le fait qu'elle soit ou pas diagonale dominante, semi-positive ou symétrique.
Les attributs dans les approches basées sur l'apprentissage permettent de déduire le meilleur algorithme sur chaque instance à résoudre. Pour construire une telle règle de déduction, on se sert d'un ensemble finis d'instances du problème. A partir de celui ci, on infère des règles de prédiction indiquant en fonction des valeurs des attributs quel est le meilleur algorithme (celui qui minimise le temps d'exécution). L'inférence des règles peut être faite en utilisant différentes techniques d'apprentissage automatique ; En particulier les arbres de décision [Bhowmick et al. 2006, Guo 2003], l'analyse statistique [Dongarra et al. 2006a, Fink 1998], l'apprentissage renforcé [Lagoudakis and Littman 2000, Kuefler and Chen 2008] ou les approches bayésiennes [Houstis et al. 2000] peuvent ici être employées.

2.3.2.1 L'apprentissage automatique dans la sélection du meilleur solveur avec PYTHIA

PYTHIA [Houstis et al. 2000] propose un mécanisme de sélection d'algorithmes pour la sélection des solveurs résolvant les équations aux dérivées partielles dans un environnement distribué. Étant donnée une équation à résoudre et en fonction d'une performance cible (précision, temps de résolution etc.), l'objectif ici est de sélectionner une paire (méthode, machine) et des paramètres numériques nécessaires à sa résolution. PYTHIA cible en particulier les équations elliptiques de la forme :

$$Lu = f \text{ sur } \omega, Bu = g \text{ sur } \Delta\omega. \tag{2.1}$$

Ici, ω est le domaine de définition et $\Delta\omega$ la frontière de celui ci.

Étant donné une instance à résoudre, la sélection des solveurs dans PYTHIA est déterminée par les attributs identifiées sur l'instance. Le processus d'inférence des règles dans PYTHIA est assez complexe et repose sur les réseaux bayésiens et de neurones [Mitchell 1997]. Les résultats obtenus avec PYTHIA dans la sélection des solveurs sont très encourageants et en font sans doute l'un des solveurs les plus efficaces pour la résolution des équations aux dérivées partielles elliptiques.

Les approches par apprentissage automatique sont parmi les approches les plus utilisées pour la combinaison automatique des algorithmes. Elles ont en particulier été employées avec succès pour construire le solveur Satzilla [Xu et al. 2008], vainqueur plusieurs fois dans l'une des plus populaires compétitions annuelles des solveurs SAT [4].
La sélection des algorithmes à partir d'une approche d'apprentissage est en général efficace lorsque les attributs déterminants dans la résolution d'un problème sont bien connus. Toutefois, ceci est difficile à garantir en théorie et en pratique [Dongarra et al. 2006b, Kuefler and Chen 2008]. Ceci est l'une des motivations des approches par portfolio d'algorithmes que nous présentons dans la suite.

2.3.3 L'approche portfolio de combinaison des algorithmes

En employant les approches basées sur l'apprentissage, on procède en associant un algorithme à toute instances à résoudre. Seulement, on peut dans certains cas ne pas prédire l'algorithme qui conduira au plus petit temps d'exécution. Pour gérer cela, on peut considérer plus d'un algorithme dans la résolution d'un problème. Seulement, il se pose dans ce cas la question de l'exécution efficace des différents algorithmes de sorte à minimiser le temps d'exécution requis pour trouver une solution. Si en effet, on exécute les algorithmes les uns après les autres, on est confronté à la définition d'un ordre d'exécution efficace dans la mesure où l'on ne connaît pas a priori l'algorithme qui en premier va donner une solution au problème. Face à cette difficulté, on peut essayer de déterminer à partir des statistiques d'exécution le meilleur ordre d'exécution. Une approche similaire a été employée dans [Bhowmick et al. 2002]. L'idée est de tester statistiquement les différents ordre d'exécution sur un ensemble finis d'instances et de choisir celui conduisant en moyenne au plus petit temps de résolution. Seulement dans ce cas, on a un nombre exponentiel de combinaison possibles et éventuellement un surcoût important sur les instances ou l'ordre d'exécution défini n'est pas efficace.
Pour exécuter efficacement les différents algorithmes sélectionnés pour résoudre une instance, nous proposons d'adopter le modèle d'exécution par portfolio [Huberman et al. 1997]
[Gomes and Selman 2001] que nous définissons dans la suite.

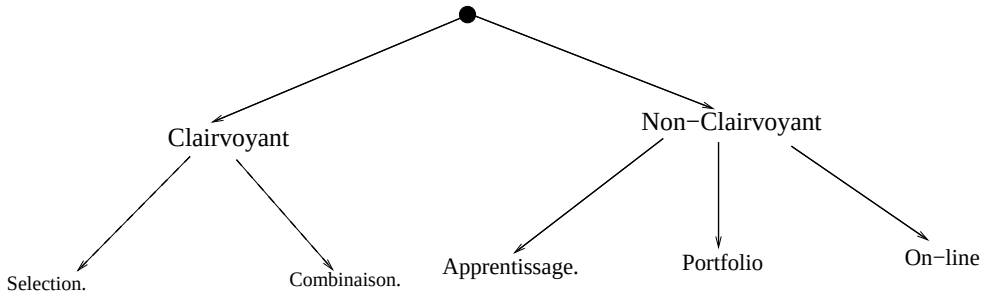


FIG. 2.2 – Vue d'ensemble des approches de combinaison d'algorithmes

[4] `http ://www.satcompetition.org`

2.3.3.1 Les portfolio d'algorithmes

Le modèle d'exécution par portfolio a été introduit dans [Huberman et al. 1997] [Gomes and Selman 2001] pour la combinaison d'algorithmes randomisés. Il est basé sur une adaptation du modèle on-line de sélection des portfolio [Markowitz 1999, Borodin and El-Yaniv 1998] aux algorithmes [5].

Supposons que l'on a deux algorithmes interruptibles A_1 et A_2 résolvant une instance I. Nous considérons aussi une unité fixe δ_t de discrétisation du temps. Pour résoudre I, à partir de la date t_0 on peut procéder en exécutant A_1 dans les intervalles de temps $[t_0 + 2\alpha\delta_t, t_0 + (2\alpha+1)\delta_t[$ et A_2 dans les intervalles $[t_0 + (2\alpha+1)\delta_t, t_0 + (2\alpha+2)\delta_t[, \alpha = 0, 1, \ldots$ jusqu'à ce qu'un algorithme trouve une solution. Si ces algorithmes sont parallèles, étant donné m ressources homogènes, on peut entrelacer leur exécution en exécutant A_1 sur m_1 ressources et A_2 sur m_2 ressources avec $m_1 + m_2 \leq m$. Ces différentes exécutions définissent des portfolio d'algorithmes. Plus formellement, nous avons la définition suivante :

Définition 2.3.1. *Supposons que nous sommes dans un contexte parallèle avec m ressources identiques discrètes. Supposons aussi que le temps est discrétisé et que la résolution d'une instance avec un algorithme prendra au plus B unités de temps. Nous définissons un portfolio d'algorithmes comme un vecteur $V = (v_1, \ldots, v_B)$ tel que chaque $v_q = (S_1^q, \ldots, S_k^q)$ avec $|\mathcal{A}| = k$, $S^q_{i,i=1,\ldots,k} \in \{0, \ldots, m\}$, $\sum_{i=1}^k S_i^q \leq m$.*

Ce vecteur définit un ordre d'exécution des algorithmes en supposant une discrétisation du temps. Chaque vecteur v_q définit un partage de ressources et a matrice V contient la succession des partages de ressources à travers les unités de temps. Supposons que δ_t est l'unité de discrétisation de temps et que l'on commence la résolution d'une instance à la date t_0.
Dans cet ordre d'exécution, si à la date $t_0 + q\delta_t$ une instance n'est pas résolue, alors pendant les dates $t_0 + q\delta_t$ et $t_0 + (q+1)\delta_t$, si $v_q = (S_1^q, \ldots, S_k^q)$, on va exécuter sur l'instance à résoudre chaque algorithme A_i sur S_i^q ressources. En particulier, si $S_i^q = 0$, alors l'algorithme A_i ne sera pas exécuté.

Une exécution par portfolio d'algorithmes généralise l'exécution des algorithmes les uns après les autres car cette dernière n'est qu'un portfolio particulier. L'avantage avec ce modèle d'exécution est de donner beaucoup plus de flexibilité au choix de la bonne combinaison d'algorithmes pour résoudre une instance.
Avec les portfolio d'algorithmes, la résolution des instances d'un problème est pensé en trois phases. Ce sont :

- La sélection du sous ensemble d'algorithmes $\mathcal{A}^I$ qui a priori sur l'instance donnera le meilleur temps d'exécution possible
- La détermination du meilleur portfolio d'algorithmes entrelaçant les exécutions des algorithmes dans $\mathcal{A}^I$
- L'exécution de ce portfolio afin de résoudre I

[5] Étant donné une séquence de prix $X = x_1, \ldots, x_n$ avec $x_j = (x_{j1}, \ldots, x_{jk})^t$, le problème on-line de sélection des portfolio consiste à choisir une séquence $B = b_1, \ldots, b_n$ telle que $b_j = (b_{1j}, \ldots, b_{kj})$ et $\sum_{i=1}^k b_{ij} = 1$ de sorte à maximiser $\sum_{j=1}^n b_j^t . x_j$

La sélection du sous ensemble d'algorithmes $\mathcal{A}^I$ peut être effectuée en utilisant une approche basée sur l'apprentissage automatique avec le sous ensemble fini $\mathcal{I}$ d'instances. Ce sous ensemble peut en outre être utilisé pour ajuster l'entrelacement des exécutions d'algorithmes.

Il existe d'autres études qui considèrent les portfolio d'algorithmes avec le temps et les ressources non discrètes [Sayag et al. 2006]. Néanmoins, nous pensons que cette hypothèse n'est pas réaliste en pratique. Le modèle d'exécution par portfolio d'algorithmes suggère un nombre important de possibilités de combinaison d'algorithmes. Afin de proposer des solutions efficaces pour gérer la combinatoire importante qu'il contient, l'on se restreint souvent à deux modèles de combinaison particuliers selon que l'on partage le temps ou les ressources.
Dans le partage de temps, on ne prend pas en compte le parallélisme dans l'exécution des instances. A chaque unité de temps, il y a donc un seul algorithme exécuté. Un modèle de partage de temps peut donc être défini comme une fonction $Ts : \{1, \ldots, B\} \longrightarrow \mathcal{A}$. Ici, $Ts(t)$ indique l'algorithme à exécuter à chaque unité de temps si à cette date l'instance en cours n'est pas résolue.
Dans le modèle de partage des ressources, l'exécution des algorithmes n'est pas entrelacée dans le temps. Ainsi, un ordre d'exécution par partage des ressources est défini par un vecteur $(S_1, \ldots, S_k)$ tel que $S_{i,i=1,\ldots,k} \in \{0, \ldots, m\}$, $\sum_{i=1}^{k} S_i \leq m$. Avec un tel portfolio, une instance est résolue en exécutant chaque algorithme A_i sur S_i ressources jusqu'à ce qu'un algorithme trouve une solution.

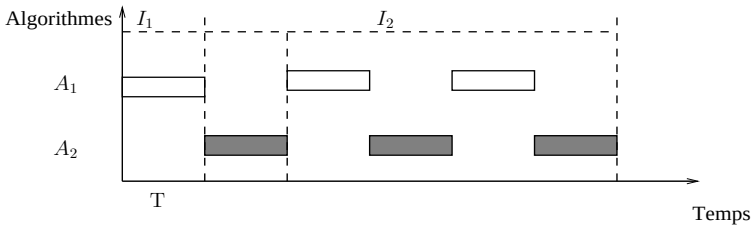


FIG. 2.3 – Exemple d'exécution par partage de temps avec deux algorithmes (A_1, A_2). Après T unités de temps ici, un algorithme est préempté.

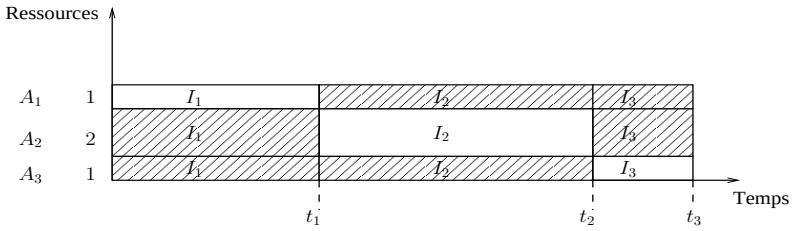


FIG. 2.4 – Exemple d'exécution par partage de ressources avec trois algorithmes et trois instances. L'exécution de I_1 se termine sur A_1 après t_1 unités de temps. L'exécution de I_2 se termine sur A_2 après $t_2 - t_1$ unités de temps. La partie hachurée correspond aux exécutions redondantes et inutiles

2.3.3.2 Intérêt des portfolio d'algorithmes

Il n'est pas évident de bien comprendre l'intérêt des portfolio d'algorithmes du fait de la redondance des calculs que l'on a ici dans la résolution de toute instance. A priori, cette approche paraît pénalisante en comparaison des approches basées uniquement sur l'apprentissage automatique dans laquelle on sélectionne un seul algorithme. Seulement, nous faisons les observations suivantes :

- En supposant que dans une approche par portfolio, on sélectionne (par le biais de l'apprentissage automatique) un ensemble d'algorithmes que l'on exécute en portfolio, on peut déterminer la meilleure exécution par portfolio en testant l'approche complète (sélection des algorithmes et exécution par portfolio) sur un jeu d'instances sur lequel on teste diverses combinaisons par portfolio d'algorithmes. Parmi les exécutions que l'on considère ainsi, il y a la situation qui correspond au choix d'un seul algorithme pour résoudre l'instance cible. L'approche par portfolio d'algorithmes dans ce cas généralise la sélection d'un seul algorithme.
- Lorsqu'il y a une grande variabilité des temps d'exécution entre les instances du problème, l'exécution concurrente des algorithmes peut être très bénéfique. En particulier considérons le jeu de données suivant : Supposons que l'on est dans un contexte parallèle avec m res-

C(.)	A_1	$\dots$	A_k
I_1	ϵ		γ
$\vdots$	γ	$\ddots$	
I_k			ϵ
I_{k+1}	ϵ	$\dots$	ϵ
$\vdots$	$\vdots$	$\ddots$	$\vdots$
I_n	ϵ	$\dots$	ϵ

TAB. 2.1 – Matrice des coûts d'exécution sur n instances avec k algorithmes.

sources et que les coûts ci dessus sont ceux obtenus par les algorithmes exécutés avec m ressources sur les instances. Supposons aussi que l'on peut déduire le coût d'exécution sur p ressources avec la relation $C(A_i, I_j, p) = \frac{mC(A_i,I_j,m)}{p}$ qui correspond à une répartition proportionnelle. Si la valeur de γ est suffisamment grande en comparaison de ϵ (par exemple $\gamma \geq \max\{nm \sum C(A_i, I_i, 1)\}$), le choix de n'importe quel algorithme A_i s'avère plus coûteux que l'exécution concurrente d'au moins deux algorithmes. Cet exemple montre que si la variabilité des temps d'exécution entre algorithmes est importante, le surcoût dans l'approche portfolio peut être plus léger que le choix d'un seul algorithme. L'exemple ci contre a été généralisé dans [Huberman et al. 1997] sur le cas des algorithmes randomisés en montrant en particulier que dans ces cas, l'exécution concurrente de plusieurs algorithmes résolvant le même problème représente un meilleur risque.

L'intérêt des portfolio d'algorithmes tient donc du fait qu'ils généralisent les approches par apprentissage automatique tout en étant particulièrement efficaces lorsqu'à travers les instances d'un problème, on a une grande variation des temps d'exécution.

2.4 Conclusion

Nous avons présenté dans ce chapitre différentes approches de combinaison des algorithmes. Les approches présentées sont rangées en deux familles : la famille des approches clairvoyantes et celle des approches non clairvoyantes. Chacune de ces approches se justifie comme nous l'avons montré sur certains types de problèmes. Pour les problèmes combinatoires difficiles en particulier ainsi que les problèmes numériques creux, les approches non clairvoyantes sont plus adéquates par exemple. Nous nous intéressons dans cette thèse à la combinaison automatique des algorithmes pour ce genre de problèmes en particulier. Parmi les approches employables ici, nous proposons d'utiliser les approches par portfolio qui peuvent être vues comme une généralisation des approches basées sur l'apprentissage avec l'avantage sur les approches on-line de prendre plus en considération le problème cible de la combinaison.

Dans la suite de cette thèse, nous allons présenter deux approches par portfolio qui peuvent être utilisées à cet effet. La première au Chapitre 3 vise à déterminer dynamiquement pour chaque instance à résoudre, la meilleure combinaison d'algorithmes pour donner une solution à chaque instance en le plus petit temps d'exécution. La seconde approche que nous présentons dans le chapitres 4 et 5 vise elle à trouver le meilleur portfolio statique d'algorithmes pour résoudre efficacement un problème étant donné un sous ensemble représentatifs d'instances de celui ci.

Chapitre 3

Construction dynamique des portfolio d'algorithmes

Résumé : *Dans ce chapitre nous présentons une approche dynamique de sélection des portfolio d'algorithmes. Nous introduisons d'abord le contexte général qui motive notre étude puis nous présentons une approche dynamique de construction des portfolio. Nous validons ensuite expérimentalement cette approche sur la résolution des systèmes linéaires creux avec des méthodes itératives.*

3.1 Introduction

Afin d'exploiter efficacement la diversité des temps d'exécution observés à travers les algorithmes dans la résolution d'un problème, les approches basées sur l'apprentissage automatique [Mitchell 1997] peuvent être utilisées. La plupart de ces approches reposent sur l'idée de trouver pour toute instance du problème à résoudre, le meilleur algorithme (celui qui minimise le temps d'exécution) parmi la liste des algorithmes candidats.

Supposons que l'on ait un problème cible $\mathcal{P}$ à résoudre avec un ensemble fini d'algorithmes candidats $\mathcal{A}$. Dans ces approches, on utilise les observations liées au comportement des algorithmes candidats sur un sous ensemble $\mathcal{I}$ d'instances de $\mathcal{P}$ pour inférer un mécanisme de choix des algorithmes. La méthodologie générale employée ici repose sur les points suivants :

1. Pour le problème $\mathcal{P}$, construire un vecteur d'attributs significatifs et évaluables permettant de caractériser ses instances. Cette phase est délicate car une bonne connaissance du problème est nécessaire. Dans la résolution d'un système linéaire par exemple, des attributs peuvent être le nombre d'éléments non nuls de la matrice, la forme de la matrice, son conditionnement, le fait qu'elle soit ou pas diagonale dominante, semi-positive ou symétrique. Un attribut est dit évaluable (ou extractible) si l'on dispose d'un algorithme capable sur une instance de calculer sa valeur.

2. Sur un ensemble fini d'instances représentatives $\mathcal{I}$, exécuter les algorithmes candidats et collecter les performances produites (en particulier, il faut collecter le temps d'exécution).

3. Appliquer *les techniques d'apprentissage automatique* sur l'ensemble d'instances $\mathcal{I}$ avec les performances d'exécutions observées pour inférer un mécanisme de prédiction **MP** qui étant données des valeurs précises du vecteur d'attributs peuvent lui associer un algorithme.

4. Étant donnée une instance du problème à résoudre, extraire les valeurs de son vecteur d'attributs et exécuter pour le résoudre l'algorithme proposé par **MP**.

Cette méthodologie a été utilisée pour implémenter plusieurs modèles de sélection dynamique d'algorithmes en employant comme technique d'apprentissage automatique notamment : les arbres de décision [Bhowmick et al. 2006], les réseaux bayésiens [Guo 2003] ou l'apprentissage statistique [Dongarra et al. 2006a].

L'usage des techniques d'apprentissage pour la sélection automatique présente toutefois deux inconvénients. Ce sont :

– La difficulté d'obtenir un *vecteur d'attributs* pour décider du meilleur algorithme.
– La non prise en compte de l'erreur potentielle commise dans la prédiction.

Le premier inconvénient s'observe facilement dans la résolution des systèmes linéaires creux avec des méthodes itératives. Ici en effet, la convergence de la méthode itérative dépend de nombreux éléments qui parfois ne peuvent être calculés qu'au prix de la résolution du système. C'est en particulier ici la distribution des valeurs propres [Saad 2003, Barrett et al. 1996]. Mieux encore, même quand théoriquement cette convergence est garantie, elle peut ne pas être effective en pratique du fait des erreurs d'arrondis [Saad 2003, Barrett et al. 1996]. Le second inconvénient est lié au fait qu'un mauvais choix d'algorithmes peut dans certaines situations se révéler très coûteux. Une telle situation est observable lorsqu'une méthode qui ne convergera jamais est sélectionnée pour la résolution d'un système linéaire alors que n'importe quelle autre méthode aurait conduit à une convergence.

Dans ce chapitre, nous proposons une approche de choix d'algorithmes inspirée de la méthodologie employée dans l'apprentissage automatique. Ici, le choix des algorithmes (en particulier les attributs) prend en compte les observations sur l'exécution des algorithmes sur un benchmark du problème cible. En outre, nous tenons compte des erreurs potentielles dans la prédiction du meilleur algorithme. Pour cela, nous considérons la possibilité de sélectionner via une approche d'apprentissage plusieurs algorithmes pour résoudre une instance que l'on combine ensuite dans une exécution concurrente par portfolio. Le résultat que nous présentons ici a été publié dans [Ngoko and Trystram 2009a]. Il présente plusieurs similitudes d'autres résultats proposés dans [O'Mahony et al. 2008, Gebruers et al. 2005] pour combiner les solveurs de satisfaction de contraintes. Les différences entre les deux approches résident essentiellement sur la stratégie de combinaison des algorithmes une fois un sous ensemble d'algorithmes sélectionnés et le fait que la sélection des algorithmes dans [Ngoko and Trystram 2009a] n'est pas fondée sur une caractérisation du problème. L'approche proposée ici s'inspire en particulier de la méthode des plus proches voisins [Mitchell 1997], de l'algorithme **GAMBLETA** [Gagliolo and Schmidhuber 2008] que nous allons présenter ainsi que des résultats obtenus sur la notion d'*anytime algorithm* [Zilberstein and Russell 1996].

Le reste du chapitre est organisé comme suit : Dans la Section 3.2, nous présentons une adaptation de la méthode des plus proches voisins pour l'apprentissage automatique dans le choix du meilleur algorithme. La Section 3.3 présente l'algorithme **GAMBLETA**. La Section 3.4 introduit la notion de profil de performance qui nous sert à caractériser l'exécution d'un algorithme sur une instance. Dans la Section 3.5.2, nous proposons un algorithme qui s'adapte sur l'instance à résoudre pour trouver la meilleure combinaison algorithmique. Dans la Section 3.7 nous appliquons l'algorithme proposé pour la résolution des systèmes linéaires et nous concluons à la section 3.8.

3.2 La méthode des plus proches voisins pour la sélection du meilleur algorithme

La méthode q-plus proches voisins [Mitchell 1997] est une méthode de classification supervisée utilisée dans l'apprentissage automatique. Ici, on range tout nouvel élément que l'on veut classer dans la classe de l'élément qui est le plus fréquent parmi ses q plus proches voisins au sens d'une distance donnée sur les attributs. Nous précisons qu'initialement ici, on suppose que l'on dispose d'une liste préalable d'éléments dont on connaît la classe.

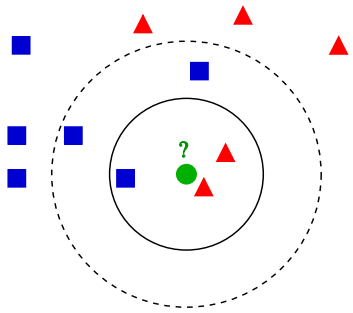


FIG. 3.1 – Illustration de la méthode des q-plus proches voisins. On veut classer la boule verte à partir des triangles et des carrés. Si $q = 3$, la boule sera mis dans la classe des triangles. Si $q = 5$, la boule est dans la classe des rectangles.

La méthode des q plus proches voisins peut être adaptée pour la sélection du meilleur algorithme en procédant comme suit :

1. On considère que l'on a initialement un ensemble fini d'instances $\mathcal{I}$ (benchmark). Sur cet ensemble, on cherche pour chaque instance quel est le meilleur algorithme parmi une liste fini d'algorithmes candidats $\mathcal{A}$
2. Lorsqu'une nouvelle instance I est à résoudre, on compare ses attributs avec ceux de $\mathcal{I}$ et on détermine l'algorithme $A_i \in \mathcal{A}$ correspondant au meilleur algorithme résolvant les instances les plus fréquentes proches de I parmi ses q plus proches voisins dans $\mathcal{I}$
3. On exécute A_i pour résoudre I.

Dans cette démarche, chaque nouvelle instance est résolue à partir de la proximité de ses attributs avec ceux d'une liste d'instances sur laquelle on connaît quel est le meilleur algorithme. Pour définir la proximité, il faut ici employer une métrique. En général, la distance euclidienne ou la distance de manhattan sont employées dans la méthode des plus proches voisins.

Une difficulté dans la méthode des plus proches voisins réside dans la détermination de la valeur q. Différentes valeurs de q en effet conduiront certainement à différentes performances en pratique. Pour déterminer la valeur de q, deux grandes approches sont souvent utilisées. Ce sont l'approche par sous échantillonage et l'approche de la validation croisée. Dans le cas du sous échantillonage, on suppose qu'on dispose d'un ensemble d'instances $\mathcal{S}$ que l'on a séparées en sous ensembles $\mathcal{T} = \mathcal{S} \setminus \mathcal{I}$ et $\mathcal{I}$. En faisant varier q, on applique la méthode des plus proches voisins avec le benchmark $\mathcal{I}$ sur $\mathcal{T}$ et on retient la valeur qui conduit au plus petit temps d'exécution en moyenne. Dans l'approche de la validation croisée, on sépare l'ensemble $\mathcal{S}$ en k (k est à fixer) partitions $\mathcal{S}_1, \ldots, \mathcal{S}_k$. En faisant varier q, on applique la méthode des plus proches voisins en prenant successivement comme benchmark un sous ensemble S_i sur le reste des instances $\mathcal{S} \setminus \mathcal{S}_i$. On retient à la fin la valeur de q ayant conduit au plus petit temps d'exécution.

La méthode des plus proches voisins a l'avantage d'être particulièrement simple en comparaison des autres techniques d'apprentissage automatique [Mitchell 1997]. Elle comporte toutefois les deux limites que nous avons évoquées au début de ce chapitre à savoir : la rigidité du mécanisme de prédiction qui ne prend pas en compte les potentielles erreurs dans le choix du meilleur algorithme et l'usage de la notion d'attributs.

3.3 GAMBLETA

Une autre modélisation pour la sélection du meilleur algorithme a été proposée dans [Gagliolo and Schmidhuber 2008] avec l'algorithme **GAMBLETA**. Dans cette modélisation, on admet que le temps est discrétisé en des unités de durée δ_t. On suppose aussi que l'on peut exécuter concurremment en parallèle les différents algorithmes sur une fraction des ressources disponibles. Dans ce modèle, une instance est résolue en faisant des choix de combinaison d'algorithmes à différentes dates $\alpha\delta_t, \alpha = 0, 1, \ldots$. Un choix de combinaison d'algorithmes indique une répartition des ressources entre les différents algorithmes que l'on exécute en parallèle pendant une durée de δ_t. On procède ainsi jusqu'à ce que l'instance soit complètement résolue. Étant donnée une instance en cours de résolution, le choix d'une répartition des ressources dans **GAMBLETA** est opéré en utilisant un modèle inspiré du problème du bandit manchot [Auer et al. 2003].

GAMBLETA utilise un portfolio d'algorithmes déterminé dynamiquement pour résoudre chaque instance d'un problème. Son inconvénient majeur est le peu d'informations en provenance du problème à résoudre que l'on emploie.

Une idée intéressante dans **GAMBLETA** est d'utiliser des exécutions en cours des différentes

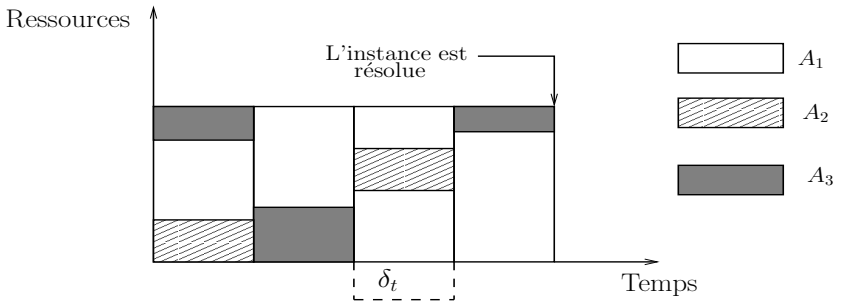


FIG. 3.2 – Exemple de résolution d'une instance avec 3 algorithmes (A_1, A_2, A_3) dans Gambleta. A chaque période de longueur δ_t, on exécute concurremment un ensemble d'algorithmes.

instances sur les algorithmes pour déterminer la bonne combinaison des algorithmes. Dans la suite, nous allons proposer une approche utilisant cette idée ainsi que les portfolio d'algorithmes dans la méthode des plus proches voisins. La notion d'attribut employée dans la méthode des plus proches voisins est difficile à déterminer étant donné un problème cible. Nous allons nous inspirer des résultats obtenus sur les algorithmes *anytime* [Zilberstein and Russell 1996] pour la définir dans notre cas. Les algorithmes *anytime* sont des algorithmes pour des problèmes d'optimisation qui à chaque unité de temps dans l'exécution peuvent proposer une approximation de la solution garantie comme étant au moins aussi bonne que les précédentes approximations. Pour ce faire, on suppose qu'étant donné le problème d'optimisation, on est capable d'associer à toute solution d'une instance une qualité. Plus précisément, étant donné une instance I, un problème d'optimisation consiste souvent à trouver une solution $S_I \in \mathcal{S}$ qui minimise ou maximise une fonction de coût $C(S_I)$. Si S_I^* est la solution optimale, on peut définir alors la qualité de la solution S_I comme étant $\frac{C(S_I)}{C(S_I^*)}$ ou $|C(S_I) - C(S_I^*)|$. Dans tous les cas, la qualité est en général considérée comme l'écart entre la solution et la solution optimale.

Nous allons dans la suite proposer une approche de combinaison d'algorithmes adaptés pour les problèmes d'optimisation. Dans celle ci, nous manipulons la notion de profil de performance que nous employons comme attributs. La section suivante explicite cette notion.

3.4 La notion de profil de performance

3.4.1 L'usage des profils de performance dans les algorithmes anytime

L'objectif du profil de performance est d'estimer la qualité d'un algorithme sur différentes instances. L'une des plus grandes difficultés dans la définition d'un profil de performance est la définition de la qualité de l'algorithme. Cette notion dépend du problème et/ou de l'algorithme

considéré [Boddy and Dean 1994, Zilberstein and Russell 1996]. Le profil de performance est en général plus facile à définir sur les problèmes d'optimisation que sur les problèmes de décision. Dans un problème d'optimisation devant minimiser une fonction de coût par exemple, la qualité d'un algorithme le résolvant peut être définie en utilisant le ratio entre le coût de la solution actuelle sur le coût connu d'une borne inférieure du problème. Ceci suppose par ailleurs que l'algorithme soit capable de fournir plusieurs solutions locales dans son exécution. Une approche sur les problèmes de décision consiste à indiquer que sur un algorithme exécuté avec une instance, le profil de performance vaut 0 tant que l'algorithme ne peut pas fournir une décision et 1 sinon [Zilberstein and Russell 1996].

Définition 3.4.1. *Soit P un problème et $\mathcal{I}$ la liste de ses instances. Le profil de performance d'un algorithme résolvant P est une fonction $PP : \mathcal{I} \times \mathbb{R}^+ \longrightarrow [0,1]$ qui pour toute instance $I \in \mathcal{I}$ et instant t indique quelle est la qualité $PP(I,t)$ de l'algorithme lorsqu'il est exécuté avec l'instance.*

Afin de donner un sens plus précis à la notion de profil de performance, on adoptera la convention suivante : Étant données une date t et une instance I quelconque dans l'exécution d'un algorithme :

$$PP(I,t) = \begin{cases} 1 & \text{si à partir de la date } t\text{, la qualité de l'algorithme ne s'améliore plus} \\ \in [0,1[& \text{Sinon} \end{cases}$$

La notion de profil de performance a beaucoup été étudiée sur les algorithmes dit *anytime* [Boddy and Dean 1994, Zilberstein and Russell 1996, Zilberstein 1996]. Plusieurs études ont ainsi été menées dans la composition des algorithmes anytime afin de produire des algorithmes de meilleur qualité. En particulier dans [Zilberstein and Russell 1996, Zilberstein 1996], la composition des algorithmes est étudiée en faisant usage de la notion de profil de performance. Dans ces études, on considère pour la composition des algorithmes, la possibilité de prédire les profils de performance ou même d'en proposer une modélisation analytique. Cette hypothèse est en outre validée expérimentalement sur plusieurs cas d'études. Si l'hypothèse d'une modélisation analytique du PP d'un algorithme a peu de fondements théoriques, ces résultats suggèrent que la notion de profil de performance peut être utilisée afin de combiner efficacement des algorithmes.

Les études faites sur la composition des algorithmes anytime ne considèrent le problème de combinaison que d'un point de vue global sans tenir compte des particularités liées aux instances du problème. Il s'agit plus ici de produire un nouvel algorithme hybride que de trouver un algorithme capable d'adapter son exécution en fonction de l'instance à résoudre. Nous pensons toutefois la notion de profil de performance peut être employée pour combiner automatiquement les algorithmes à partir des instances d'un problème. Dans la suite, nous montrerons comment la notion de profil de performance peut être utilisée comme alternative à la notion d'attributs.

3.4.2 Le profil de performance comme attributs

Nous adopterons dans la suite les profils de performance comme attributs. La notion de profil de performance n'est pas toujours simple à définir étant donné un problème cible. Toutefois, on peut noter que plusieurs algorithmes dans les problèmes d'optimisation (ordonnancement, SAT etc.) et numériques (résolution des équations etc.) on trouve dans plusieurs cas des algorithmes qui procèdent en calculant successivement des solutions locales. En outre, il est en général possible de comparer cette solution locale avec une borne inférieure sur la meilleure solution possible sur les problèmes combinatoires. Ceci peut donc nous servir pour définir la qualité de la solution à chaque instant. Dans le cas des problèmes numériques, on peut pour la résolution des équations comparer les approximations successives ou estimer le résidu dans le cas des systèmes linéaires pour avoir une idée de la qualité de la solution qu'on a.
Il est donc possible d'obtenir des définitions du profil de performances notamment sur les problèmes combinatoires et numériques en prenant à différents instants de temps la qualité de la solution obtenue sur l'algorithme que l'on exécute. Dans la suite, nous allons voir comment employer cette notion pour la combinaison des algorithmes. L'usage des profils de performance comme attributs est proche de la technique du *lookahead* [Coleman 2006] sur les algorithmes on-line. Sur les algorithmes d'ordonnancement de tâches dans un contexte on-line l'usage du lookahead suppose que l'on puisse par exemple connaître lorsque l'on place une tâche un sous ensemble des tâches à placer ultérieurement. La similitude avec le profil de performance vient du fait qu'en l'utilisant, on suppose pour sélectionner un algorithme que l'on connaît une partie de son comportement futur.

3.5 Sélection dynamique des portfolio

L'algorithme que nous proposons dans cette partie (**ACPP**) est basé sur les profils de performance. Il s'inspire de la méthode des plus proches voisins dans lequel on a introduit les portfolio d'algorithmes et les profils de performance. Nous supposons ici que les algorithmes candidats sont interruptibles et que l'on a l'hypothèse de similitude. Le principe de **ACPP** est donné ci dessous.

3.5.1 Principe

La méthode des q plus proches voisins pour la sélection des algorithmes décrite à la section 3.2 est limitée par la rigidité du mécanisme de prédiction et la notion d'attributs.

Pour rendre le mécanisme de prédiction plus flexible, nous proposons ici d'exécuter concurremment tous les algorithmes candidats correspondants aux q plus proches voisins connus pour l'instance à résoudre. Ainsi, nous réduisons potentiellement le surcoût que l'on peut obtenir si le plus proche voisin identifié correspond en fait à un algorithme candidat très lent sur l'instance à résoudre. Seulement, cette réduction entraîne nécessairement un autre surcoût lié à la redon-

dance des calculs nécessaires à la résolution de toute instance. Toutefois, nous pensons que la sélection des algorithmes a son utilité lorsque, les différences de performance entre algorithmes sont importantes. Dans ces situations donc, un mauvais choix d'algorithme peut facilement se révéler plus coûteux que les calculs redondants.

Nous considérons dans cette méthode les attributs comme étant le profil de performance. Dès lors, à la comparaison de vecteurs d'attributs, nous substituons la comparaison des profils de performance.
L'usage du profil de performance se heurte toutefois à la difficulté de les comparer mais aussi de les déterminer pour la nouvelle instance à résoudre. Pour cela, nous supposons que le temps est discrétisé en unité de taille δ_t. Cette hypothèse nous permet de considérer les profils de performance comme des vecteurs donnant une valeur à chaque unité de temps δ_t. On peut alors les comparer en utilisant par exemple la distance euclidienne. Étant donnée une instance à résoudre, il est difficile d'avoir son profil de performance. Pour en avoir une idée, nous proposons de fixer une période de T unités de temps et déterminer son profil sur celle ci. Pour cela, nous exécutons l'instance à résoudre avec les différents algorithmes pendant T unités de temps.

Une autre difficulté se présente lorsque nous avons déterminé les différents algorithmes à exécuter sur l'instance à résoudre. Ne sachant pas quel est l'algorithme candidat qui va le plus rapidement résoudre l'instance, il faut adopter une stratégie intelligente d'exécution afin d'obtenir un résultat le plus rapidement possible. Nous proposons ici d'adopter un portfolio d'algorithmes que nous appelons le portfolio de l'allocation moyenne (**MA**). Supposons que l'on a k algorithmes candidats $\{A_1, \ldots, A_k\}$. Dans **MA** Si on commence l'exécution des algorithmes à la date t_0 alors entre chaque date $t_0 + \alpha\delta_t$ et $t_0 + (\alpha + 1)\delta_t$, on exécute l'algorithme $A_{\alpha \bmod k + 1}$.
L'intérêt de **MA** est de tenir compte du fait que l'on ne sait pas a priori quel est l'algorithme qui va en premier lieu résoudre l'instance. Son exécution suppose que l'on peut interrompre les algorithmes sur l'unité de temps que l'on s'est fixée.

Une difficulté importante dans le principe ci dessus est dans le choix de la valeur de T. La détermination de cette valeur pose en effet un dilemme exploration/exploitation car intuitivement, de grandes valeurs de T permettent d'avoir des profils de performance très précis. Néanmoins, ceci entraîne un surcoût lié à l'exécution des algorithmes de tous les algorithmes.
Dans la suite, nous présentons l'algorithme **ACPP** bâti sur le principe ci dessus.

3.5.2 L'algorithme ACPP

L'algorithme **ACPP** procède comme suit :

1. Étant donné un benchmark d'instances $\mathcal{I}$ et un ensemble d'algorithmes $\mathcal{A}$, on construit le profil de performance $PP : \mathcal{I} \times \mathcal{A} \longrightarrow [0, 1]^T$ des algorithmes candidats de $\mathcal{A}$ sur $\mathcal{I}$ sur une durée de T unités de temps.
2. On construit la fonction $Best : \mathcal{I} \longrightarrow \mathcal{A}$ qui indique pour chaque instance du benchmark quel est le meilleur algorithme pour lui (celui qui le résout en un temps d'exécution minimal).

3. Étant donnée une instance I à résoudre, on l'exécute sur les algorithmes dans $\mathcal{A}$ durant T unités de temps pour chaque algorithme. On collecte ensuite les profils de performance de cette instance $PP_i^I = [PP_i^I(1), \ldots, PP_i^I(T)]^T$ pour chaque algorithme A_i pendant les périodes de longueur δ_t.
4. On compare $PP_i^I, \forall i$ avec $PP(I_j, A_i)$ et on sélectionne les q instances du benchmark($\mathcal{I}_s$) qui sont proches de I.
5. On continue l'exécution de I en utilisant l'algorithme de l'allocation moyenne **MA** avec les meilleurs algorithmes (déduit de la fonction $Best$) sur les instances dans $\mathcal{I}_s$.

Les q instances les plus proches sont sélectionnées en prenant les q premières instances I_j minimisant $v(i,j) = (\|PP_i^I - PP(I_j, A_i)\|_2)$ avec $1 \leq i \leq k$ et $1 \leq j \leq n$ (n est ici le nombre d'instances du benchmark). L'algorithme **ACPP** comporte deux phases. Une phase

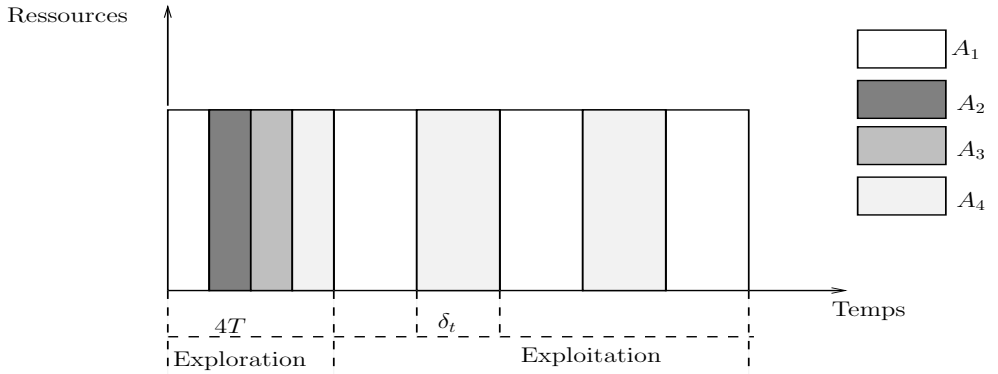


FIG. 3.3 – Illustration des phases de l'algorithme ACCP avec 4 algorithmes. On collecte les profils de performance sur une durée de $4T$. On sélectionne ensuite deux algorithmes en se basant sur ces profils et on les exécute concurremment jusqu'à ce que l'un résolve l'instance.

d'exploration où pour l'instance à résoudre on collecte les profils de performance et on établit la liste des algorithmes qui doivent rapidement résoudre l'instance et une phase d'exploitation. Dans cet algorithme, les paramètres T et q doivent être fixés au préalable. Nous allons voir comment les fixer dans la suite.

3.5.2.1 Détermination de T et q

Pour fixer les valeurs de T et q, nous adoptons la méthode du sous échantillonnage que nous avons présenté à la section 3.2. Aussi, on suppose initialement qu'on a un ensemble d'instances $\mathcal{T}$ que l'on doit utiliser pour la validation avec $\mathcal{T} \cap \mathcal{I} = \emptyset$. Nous supposons aussi qu'on exécute chaque algorithme sur une instance en au plus B unités de temps. En faisant varier T de 1 à B et q de 1 à n nous déterminons à chaque fois le temps moyen d'exécution qu'on met en appliquant l'algorithme **ACPP**. A la fin nous prenons les valeurs de T et de q sur lesquels nous obtenons le plus petit temps d'exécution.

3.5.3 Complexité

Dans cette partie, nous analysons la complexité de la prédiction avec l'algorithme **ACPP** ainsi que celle de l'estimation de ses paramètres en utilisant la technique du sous échantillonage. Nous adoptons les notations suivantes : $|\mathcal{I}| = n$, $|\mathcal{A}| = k$ et $|\mathcal{T}| = m$. Nous supposons que $C_{MA}(q, x)$ est une borne supérieure sur le temps d'exécution de MA lorsqu'il est exécuté avec q algorithmes sur une instance qui a besoin d'au plus x unités de temps pour être résolue par un des algorithmes. En négligeant les surcoûts liés à la préemption, il est facile d'établir que $C_{MA}(q, x) \leq qx$. Nous supposons aussi que l'exécution d'un algorithme dans $\mathcal{A}$ pendant une période de δ_t ainsi que la collecte de son profil de performance est en $O(1)$. Nous avons le résultat suivant :

Théorème 3.5.1. *Étant donnée une instance quelconque, l'exploration sur cette instance dans* ***ACPP*** *peut se faire en* $O(n(\log n + kT))$ *et le sous échantillonage pour déterminer* q *et* T *en* $O(mnB(n \log n + nk\frac{B+1}{2} + C_{MA}(k, B)))$

Démonstration. Dans l'exploration, on a trois étapes. Une étape d'exécution des algorithmes et de collecte des profils de performance, une étape de comparaison des profils de performance et une étape de sélection des algorithmes à exécuter dans la phase d'exploitation.
D'après les hypothèses précédentes, la première étape est en $O(kT)$ car on exécute k algorithmes chacun sur une période de T unités de temps. La comparaison des profils de performance dépend du calcul des valeurs $v(j)$ pour $I_j \in \mathcal{I}$. Le calcul d'une valeur $\|PP_i^I - PP(I_j, A_i)\|_2$ peut facilement être fait en $O(T)$ étant donné que l'on a des vecteurs de taille T. Aussi, le calcul de toutes les valeurs $v(j)$, pour tout I_j peut être fait en $O(nkT)$. Dans la troisième étape, l'on doit sélectionner les algorithmes des q instances ayant les plus petites valeurs $v(j)$, on peut procéder en triant le vecteur v ce qui peut être fait en $O(n \log n)$. Ainsi, la complexité de la phase d'exploration est en $O(n(\log n + kT))$.
Dans la méthode du sous échantillonage, on exécute pour chaque valeur $q, 1 \leq q \leq n$ et $T, 1 \leq T \leq B$ l'algorithme **ACPP** sur les instances dans $\mathcal{T}$. L'exécution d'une instance dans $\mathcal{T}$ peut être faite en $O(n(\log n + kT)) + O(C_{MA}(k, B))$. L'exécution des instances dans $\mathcal{T}$ étant donné une valeur (q, T) peut donc être faite en $O(m(n \log n + nkT + C_{MA}(k, B)))$. Aussi toutes les exécutions peuvent être faites en $\sum_{q=1}^{n} \sum_{T=1}^{B} O(m(n \log n + nkT + C_{MA}(k, B))) = O(mnB(n \log n + nk\frac{B+1}{2} + C_{MA}(k, B)))$. En supposant qu'à la fin de l'exécution des instances dans $\mathcal{T}$ on peut obtenir le temps moyen d'exécution en $O(m)$, on a le résultat. ❑

Ce résultat montre que la taille du benchmark ainsi que la longueur de la phase d'exploration sont importants dans l'exploration avec **ACPP**. Il montre aussi que si $C_{MA}(k, B)$ est une fonction polynomiale de ses paramètres, alors l'on peut déterminer les paramètres de **ACPP** en temps polynomial.

Plusieurs variantes peuvent être proposées pour l'algorithme **ACPP**. Nous allons présenter quelques unes d'entre elles dans la section suivante.

3.6 Autre variantes de l'algorithme ACPP

Plusieurs variantes peuvent être proposées dans l'algorithme **ACPP** en modifiant certaines partie de son exécution. Nous allons ici présenter trois modifications possibles. Ce sont :

1. L'usage des profils de performance multidimensionnels
2. L'usage d'un modèle de portfolio d'algorithmes par partage de ressources

Nous avons jusqu'ici supposé qu'étant donnés une instance et un algorithme, on avait un profil d'exécution. Afin d'être plus précis, on peut adopter plusieurs profils d'exécution pour caractériser l'exécution d'un algorithme sur une instance. Dans cette situation le calcul de $v(j)$ doit tenir compte de tous les profils d'exécution. Cela peut être fait en appliquant la norme F sur les profils de performance.

Dans un contexte parallèle, on peut trouver une alternative à l'algorithme de l'allocation moyenne que nous avons proposé. On suppose que l'on a p ressources et k algorithmes, et que l'on a la décomposition *euclidienne* $m = qk + r$ (où $0 \leq r < k$). On alloue $q + 1$ ressources à r algorithmes $A_i, 1 \leq i \leq k$ et q ressources aux $k - r$ restant. Sur une instance à résoudre, on exécute alors tous les algorithmes sur les ressources alloués et on s'arrête dès qu'un algorithme trouve une solution.

Dans la suite nous proposons des expérimentations réalisées avec **ACPP** pour la résolution des systèmes linéaires.

3.7 Application à la résolution des systèmes d'équations linéaires

3.7.1 Problématique et algorithmes

Nous considérons les systèmes linéaires d'équations de la forme $Ax = b$ où $A \in R^{p \times p}, x, b \in R^p$. Un tel système est creux si le ratio des valeurs nulles sur les valeurs non nulles dans la matrice A est important. Dans la résolution des grands systèmes linéaires creux, les méthodes itératives sont recommandées pour le bon compromis qu'elles offrent entre le temps de résolution de l'équation et la qualité de la solution obtenue. Plusieurs méthodes itératives ont été développées dans la résolution des systèmes linéaires avec en particulier la classes des méthodes de la base de Krylov. Sur ces méthodes, en fonction de certaines propriétés théoriques de la matrice A (symétrique, définie positive ...), l'on peut avoir différentes vitesses de convergence [Saad 2003, Nachtigal et al. 1992, Ern et al. 1994]. Néanmoins, il reste difficile de vérifier ces propriétés en pratique de sorte à les utiliser pour déterminer étant donné un système quelle est la méthode la plus adéquate [Saad 2003, Barrett et al. 1996, Bhowmick et al. 2006].

L'intérêt de combiner différents algorithmes issus des méthodes itératives est motivé en particulier par la résolution des systèmes non linéaires. En effet, la résolution des systèmes non linéaires fait en général usage des algorithmes itératifs non linéaires qui au fil des itérations génèrent des systèmes linéaires [Kelley 1995]. Les propriétés de ces systèmes par ailleurs peuvent changer au cours des itérations non linéaires de sorte qu'une méthode efficace à une itération ne le soit plus sur les itérations suivantes [Ern et al. 1994].

Nous allons dans cette partie appliquer la technique **ACPP** afin de trouver un meilleur compromis dans le temps de résolution d'un ensemble de systèmes d'équations linéaires avec un ensemble finis de méthodes itératives. Les méthodes que nous considérons ici sont :

1. La méthode du gradient conjugué carré (CGS)
2. La méthode du gradient biconjugué (BICG)
3. La méthode du gradient biconjugué stabilisé (BICGSTAB)
4. La méthode des résidus minimaux transposé (TFQMR)

Toutes ces méthodes sont de la base de Krylov [Saad 2003]. Nous les avons sélectionnées en raison des complémentarités observées entre elles et soulignées dans [Barrett et al. 1996]. Nous adoptons pour ces méthodes les implantations en langage Matlab proposées dans [Barrett et al. 1994]. Nous avons aussi envisagé l'utilisation de la méthode des résidus minimaux générale (GMRES). Nous nous sommes toutefois heurtés dans ce cas à la difficulté de définir efficacement la préemption à cause du redémarrage interne nécessaire pour limiter la capacité mémoire dans cette méthode.

L'utilisation d'**ACPP** pour combiner des systèmes linéaires se heurte à la difficulté qui est la justification de la notion de benchmark. Ceci est dû au fait que les systèmes creux sont caractérisés par la grande diversité possible des structures de la matrice A. Toutefois, nous remarquons que dans la pratique, la plupart des systèmes creux sont souvent issus des domaines d'applications comme l'astrophysique, l'ingénierie chimique, la modélisation océanique etc. Pour ces domaines d'applications, de nombreuses collections de matrices prenant en compte les structures de matrices les plus observées sont disponibles [1] [2]. Ces collections peuvent donc servir de benchmark.

3.7.2 Jeu de données et profil de performance

Le jeu de données que nous utilisons ici provient d'une collection de matrices disponibles à l'université de Floride [3]. Dans cette collection, nous avons extrait 959 matrices réelles et carrées dont certaines sont symétriques, définies positives, ou quelconques. Les matrices sélectionnées sont de dimension au plus 6000. Ces matrices sont issues de différentes collections faisant intervenir différents domaines comme la mécanique, l'électronique, l'ingénierie chimique etc. Nous avons utilisé ces matrices dans la résolution des systèmes $Ax = b$ avec $b = [1, \ldots, 1]^t$. avec comme approximation initiale le vecteur nul. Avec cette considération, une instance du

[1] http ://math.nist.gov/MatrixMarket/
[2] http ://www.cise.ufl.edu/research/sparse/matrices/
[3] http ://www.cise.ufl.edu/research/sparse/matrices/

problème de résolution des systèmes linéaires est simplement donnée par une matrice. Si r est le vecteur résidu obtenu à une étape, tol une tolérance que l'on se donne, le critère d'arrêt pour tous les algorithmes itératifs dans nos implantations est $\|r\| \leq tol * \|b\|$ (ici $tol = 10^{-6}$). Nous avons exécuté les différents algorithmes sur au plus 700 itérations.

Il n'est pas aisé de définir un profil de performance adéquat afin de comparer la résolution de nos systèmes avec les algorithmes itératifs. Pour le faire, nous observons d'abord qu'il n'est pas forcément pertinent de le définir par rapport au temps d'exécution dès lors que les informations qui peuvent nous permettre de jauger la qualité des algorithmes itératifs sont générés par itération (la durée des itérations n'est pas la même pour tous les algorithmes). Intéressons nous maintenant aux informations générés tout au long de l'exécution. Celles ci sont de deux sortes principalement : les vecteurs de la base de Krylov et les approximations calculées. Les bases de Krylov ne sont pas intéressantes pour la comparaison des exécutions du fait qu'elles sont potentiellement volumineuses. L'approximation calculée toutefois est intéressante car avec le résidu [4], elle permet de nous situer dans la résolution du système.
Considérons maintenant deux matrices qui auraient des propriétés similaires. Notre but est de définir pour elles un profil de performance adéquat qui permette de les comparer. Pour cela nous considérons l'exécution des algorithmes sur ces matrices. L'approximation initiale tout comme le résidu étant lié à la structure à la matrice initiale, les résidus initiaux pour les différents algorithmes sur ces matrices seront différents. Après l'exécution de $w, w > 1$ itérations l'on aura une nouvelle approximation pour chacune de ces matrices. Si les deux matrices sont similaires toutefois, nous pensons que la progression des résidus générés dans la fenêtre des w itérations doivent être comparables.
A partir de ces observations nous proposons deux définitions pour le profil de performance : Dans la première définition, étant donnée une matrice A utilisée avec un algorithme itératif, si $r^A(z)$ est le résidu obtenu en résolvant $Ax = b$ à l'itération z, nous prenons $PP^A(z) = \frac{\|r^A(z)\|}{\|r^A(z-w)\|}$ avec $w > 1$ que nous fixons. Dans la seconde définition nous prenons $PP^A(z) = \frac{\min \|r^A(z')\|}{\max \|r^A(z')\|}$ avec $z - w \leq z' \leq z$

Dans la première définition, $PP^A(z)$ donne l'évolution relative du *de la convergence* entre une fenêtre de w itérations. La seconde définition vise a prendre un peu plus en compte les différents résidus obtenus dans la fenêtre d'itérations. Dans la suite, lorsque nous désignerons des profils de la forme $PP^A(z) = \frac{\|r^A(z)\|}{\|r^A(z-w)\|}$, nous dirons que nous utilisons des profils de type 1. Lorsque nous utilisons $PP^A(z) = \frac{\min \|r^A(z')\|}{\max \|r^A(z')\|}$, nous parlerons de profil de type 2.
L'on peut noter que dans ces définitions, le profil de performance sera très près de 1 lorsque l'algorithme sera près de converger ou stagnera car dans ces cas, le résidu n'évolue plus. Nous tiendrons aussi compte de cette information en considérant en plus de la comparaison des profils de performance les valeurs de celui ci pour identifier les cas de stagnation ou de convergence à priori après quelques étapes de résolution du système.

[4] Étant donnée une approximation x^e, le résidu obtenu à partir de celle ci est égal à $\|b - Ax^e\|$

3.7.3 Plan d'expérimentation

Nous avons effectué deux séries d'expérimentations en utilisant le logiciel Octave [5] qui nous offre plusieurs facilités dans l'écriture des solveurs numériques. Les objectifs que nous poursuivons étaient : la prédiction du meilleur solveur dans la phase d'exploitation et la minimisation du nombre d'itérations dans la résolution d'un système. Nous précisons que ce dernier objectif ne correspond pas toujours à la minimisation du temps d'exécution. Nous l'avons choisi à cause de la facilité qu'elle offre dans l'implantation. Nous ajoutons qu'étant donné que nous n'avons employé que des solveurs de la famille du gradient conjugué, ces deux objectifs se rapprochent dans la mesure où les différences entre coûts d'itérations de nos solveurs ne sont pas grandes.

1. Dans la première série, nous avons évalué la capacité d'**ACPP** à prédire le meilleur algorithme sur un ensemble de matrices issus des perturbations de notre benchmark. Dans cette série, l'ensemble des 959 matrices nous a servi de benchmark. Pour chaque matrice du benchmark A, du benchmark, nous avons généré une matrice $A' = A + \alpha I$ avec α qui est une constante. L'intérêt de cette perturbation est que les distributions de valeurs propres dans A et dans A' sont proches. Étant donné que la convergence des méthodes de Krylov dépend de la distribution des valeurs propres, nous pensons donc que la meilleure méthode sur A devrait a priori être la meilleure sur A'. Dans ce cas, si le procédé de prédiction est efficace, alors avec la connaissance des matrices A dans son benchmark et du meilleur algorithme pour ces matrices, il devrait prédire efficacement le meilleur algorithme sur A'. Nous avons aussi considéré dans cette série les les matrices A' de la forme $A' = I - QDQ^t$ où Q est la matrice des vecteurs propres de A et D celle des valeurs propres (dans Matlab et Octave [Q,D] = eig(A)). Les matrices sélectionnées dans ce cas sont toutes symmétriques.

2. Dans la seconde série d'expérimentations, nous avons dans un premier temps utilisé la technique du sous échantillonage pour déterminer les valeurs de q et T. Dans celle ci nous testons **ACPP** avec différentes valeurs de q avec pour objectif de minimiser le nombre d'itérations dans la résolution d'un système (ainsi, T désigne ici le nombre d'itérations minimal dans la phase d'exploration). Pour déterminer T et q, nous avons découpé notre ensemble de matrices en deux sous ensembles de 478 et 479 matrices choisies aléatoirement. Toutefois, nous rappelons que nos matrices appartiennent à différentes collections. Dans la sélection aléatoire des matrices, nous avons garanti qu'au moins une matrice soit présente pour chaque collection. Une fois déterminées les valeurs de q et T, nous avons appliqué **ACPP** en prenant aléatoirement comme benchmark des proportions de 50%, 33%, 25% et 20% des matrices. Sur chaque proportion, nous avons effectué 50 tirages distincts. Comme précédemment, au moins une matrice dans chaque collection devait être présente dans le benchmark [6].

[5] `http://octave.org`

[6] Nous avons 55 collections de matrices

3.7.4 Première série d'expérimentations

Dans cette série, nous avons fixé $T = 10$ et $q = 1$ dans l'algorithme **ACPP** 3.5.2. Dans le cas des profils de type 1, nous fixons $w = 1$ et pour les profils de type 2, $w = 2$. Dans la table 3.1, nous donnons la proportion des solveurs que nous avons exactement prédit en utilisant ACPP sur les perturbations de la forme $A' = A + \alpha I$ en ayant utiliser les matrices A comme benchmark. Les proportions que nous obtenons sont intéressantes. Néanmoins, nous pensons

	Profil Type 1	Profil Type 2
$\alpha = 0.1$	0.95	0.92
$\alpha = 0.5$	0.93	0.93
$\alpha = 2$	0.93	0.93

TAB. 3.1 – Proportion des solveurs exactement prédit

que les résultats sont aussi influencés par le fait que dans plus de 73% des cas, nous avions un seul solveur (le BICGSTAB) qui dominait tous les autres. En particulier, lorsqu'un système n'était pas résolu par une des méthodes, nous avons considéré que sur ce système, la meilleure méthode est le BICGSTAB. Nous avons opéré cette affectation en notant que sur les systèmes résolus, le BICGSTAB dominait les autres solveurs.

Afin d'avoir une vue plus détaillée du taux de prédiction, il faut le considérer par solveurs. Dans ce cas, nous obtenons encore des résultats intéressants. Dans la table 3.2, nous présentons pour le cas des profils de type 1 la fiabilité de la prédiction. Ce résultat montre que sur les sol-

	BICGSTAB	BICG	CGS	TFQMR
$\alpha = 0.1$	1	0.80	0.82	0
$\alpha = 0.5$	0.99	0.83	0.80	0
$\alpha = 2$	0.98	0.85	0.79	0

TAB. 3.2 – Taux de prédiction par solveur dans le cas des profils de type 1

veurs BICGSTAB, BICG et CGS, nous avons obtenu de bonnes prédictions. Nous remarquons aussi que le taux de prédiction de la méthode TFQMR est très faible. Ce taux nous le pensons est influencé par le fait qu'il y avait trop peu de matrices sur lesquelles cette méthode était la meilleure dans le benchmark. Avec par exemple $\alpha = 0.1$, cette méthode était la meilleure sur seulement deux matrices. Cette observation nous indique que la qualité de la technique **ACPP** est dépendante du benchmark considéré.

Dans cette première série, nous avons aussi testé la qualité de la prédiction sur des matrices de la forme $A' = I - QDQ^t$. Lors des exécutions toutefois sur Octave, nous avons noté que le calcul de la décomposition $[Q, D] = eig(A)$ était dans certains cas très coûteux. Aussi, nous nous sommes restreint à un jeu de 47 matrices tirées aléatoirement parmi les 959 que nous avons perturbés. Avec cette perturbation nous étions capable de prédire le meilleur solveur dans 89% des cas.

Les résultats que nous avons ainsi obtenus nous montre que l'approche **ACPP** permet d'obtenir avec des taux de prédiction important le meilleur solveur.

L'ensemble des codes que nous avons utilisés dans ces expérimentations ainsi que les jeux de données sont disponibles à l'adresse : `http://moais.imag.fr/membres/yanik.ngoko/`

3.7.5 Seconde série d'expérimentations

Nous avons appliqué la technique du sous échantillonnage décrite ci dessus pour déterminer les valeurs optimales de T et q en faisant varier q de 1 à 7 et T de 7 à 20. Après ces tests nous avons obtenu comme valeurs optimales permettant de minimiser le nombre d'itérations : $q = 2$ et $T = 10$. Nous précisons que la valeur $q = 3$ a donné les mêmes performances que $q = 2$ pour $T = 10$. Nous avons retenu la valeur $q = 2$ car l'on peut noter que si à l'issue de la phase d'exploration, nous avons plusieurs voisins alors potentiellement on aura une exécution concurrente des solveurs. Plus on a de voisins, plus le nombre de solveurs intervenant dans l'exécution concurrente peut être important. Aussi, il est préférable de retenir la valeur conduisant à un plus petit nombre de voisins.

Sur la figure 3.4, nous présentons les taux de succès dans la prédiction que nous avons en appliquant **ACPP** avec $q = 2$, et $T = 10$ en prenant une proportion des instances comme benchmark. Les différents benchmarks utilisés et l'ensemble des matrices sont accessibles à l'adresse `http://moais.imag.fr/membres/yanik.ngoko/` . Étant donné que $q = 2$, il est possible que dans la phase d'exploitation dans **ACPP**, on exécute deux algorithmes dans un modèle de partage de temps. Aussi, les taux que nous présentons indiquent la proportion des succès que nous avons eu en prédisant le meilleur solveur dans un sous ensemble d'au plus deux solveurs sur les quatre utilisés.

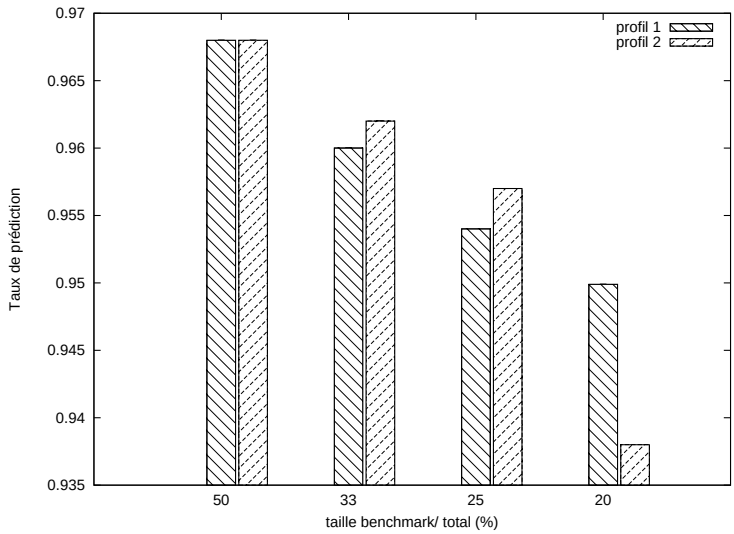


FIG. 3.4 – Taux de prédiction en fonction de la taille du benchmark

On peut noter que plus le benchmark est grand, plus le taux de prédiction l'est aussi. Ceci est tout à fait logique dans l'esprit de la méthode des plus proches voisins. On peut aussi remarquer que les deux types de profils de performance sont efficaces dans la prédiction. Sur la figure 3.5, nous présentons le rapport entre le nombre d'itérations obtenus avec **ACPP**, l'allocation moyenne et ceux de la solution optimale. On peut noter qu'en moyenne dans **ACPP**, on a un meilleur nombre d'itérations que lorsque l'on utilise l'allocation moyenne. La proximité entre le nombre des itérations avec **ACPP** et la solution optimale s'explique aussi par le fait que dans plusieurs cas, des solveurs ont interrompu leur exécution avant même d'atteindre le nombre maximum d'itérations. Ainsi, la redondance des calculs dans la stratégie de portfolio s'en trouve ainsi amortie.

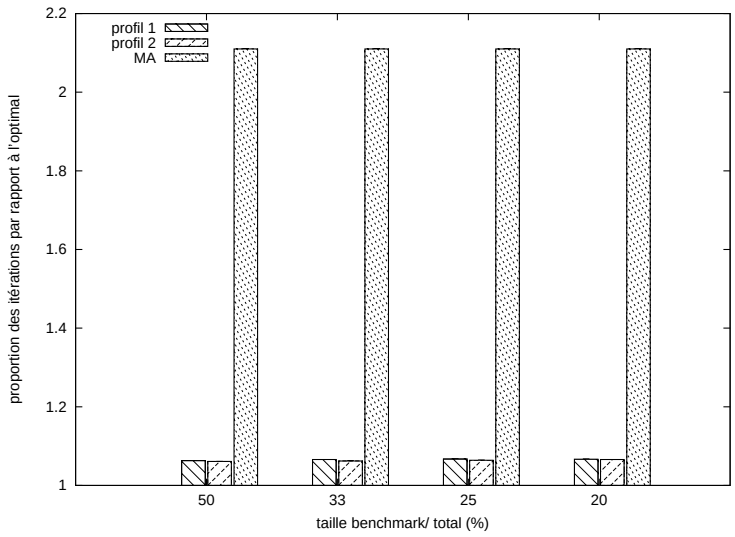


FIG. 3.5 – Proportion des itérations par rapport à la solution optimale

Le tableau 3.7.5 donne la moyenne des temps d'exécutions de la technique **ACPP**, de l'algorithme de l'allocation moyenne et du meilleur algorithme par instances. Sur les instances non résolues, nous avons pris le meilleur algorithme comme étant celui qui se termine le plus rapidement. Dans ces exécutions, nous avons pris $\delta_t = 10$ itérations pour l'allocation moyenne. Ces résultats ont été obtenus sur un processeur Dual CPU E2160 @ 1.80GHz. Les temps que nous présentons sont obtenus avec le profil de type 1. Sur le profil de type 2, nous avons des temps d'exécution similaires. Nous pouvons noter ici que la technique **ACPP** est plus coûteuse que le

Meilleur Alg par instances	MA	ACPP(50%)	ACPP(33%)	ACPP(25%)	ACPP(20%)
501	1071.77	1523.03	1243.9	943.13	886.5

TAB. 3.3 – Temps d'exécution avec le profil de type 1.

meilleur algorithme à chaque fois. Le temps d'exécution augmente avec la taille du benchmark comme l'a suggéré l'analyse théorique de la complexité. Dans les temps que nous présentons, il faut néanmoins prendre aussi en compte le temps nécessaire à la génération des profils de

performance. En pratique, on peut en effet les générer une seule fois pour multiples exécutions. Néanmoins, le temps de la technique **ACPP** reste important. On peut aussi noter que lorsque la taille du benchmark est très grande, le temps avec **ACPP** est plus grand que celui de l'allocation moyenne. Ceci en particulier parce que dans ces cas, la détermination des plus proches voisins est coûteuse. Avec **ACPP(20%)** on a un ratio sur le temps d'exécution de 1.77 avec l'algorithme optimal sur chacune des instances. Ce ratio est intéressant car nous partons sur une base de 4 algorithmes sans que le temps optimal de résolution à chaque fois ne soit multiplié en fin de compte par 4. On peut néanmoins noter que le rapport entre les temps d'exécution avec **ACPP** et l'algorithme optimal à chaque fois est plus important que le rapport entre le nombre d'itérations avec **ACPP** et ceux de la solution optimal. Avec **ACPP(20%)** par exemple, on a un rapport en moyenne près de 1.10 avec les itérations de la solution optimale. Nous expliquons cette différence entre autre par les surcoûts induit par la préemption des algorithmes dans certains cas (si l'on a deux algorithmes dans la phase d'exploitation) ainsi que le temps nécessaire à la recherche des voisins (qui n'est pas comptabilisé dans le calcul du nombre d'itérations). Une autre remarque importante que nous avons eu dans les expérimentations est que le temps d'exécution est largement influencé par la valeur de δ_t. Il est important d'élaborer une stratégie efficace pour la déterminer car si cette valeur est petite, le surcoût induit au changement de contexte est très important et si elle est grande, on a moins de marges de manoeuvre dans l'exécution concurrente.

3.8 Conclusion

Nous avons présenté dans ce chapitre une approche de combinaison des algorithmes afin d'obtenir en moyenne de meilleures performances. Cette approche s'appuie sur une technique d'apprentissage automatique dans laquelle nous rajoutons l'idée des portfolio d'algorithmes. L'approche proposée procède par une adaptation à l'instance pour déterminer le meilleur portfolio d'algorithme permettant de la résoudre. Nous l'avons analysée puis validée expérimentalement sur la résolution des systèmes linéaires afin de minimiser le nombre d'itérations nécessaires à la résolution des systèmes. Les résultats expérimentaux montrent que la technique proposée est efficace dans la prédiction du meilleur solveur ou d'un sous ensemble le contenant mais est coûteuse en temps quand la taille du benchmark est importante. Nous envisageons les perspectives suivantes à l'issue de ce travail :

1. La prise en compte de la qualité de la solution
2. L'amélioration du choix du benchmark
3. L'introduction d'une phase d'exploration progresssive.
4. L'utilisation d'un modèle mathématique de qualité
5. Le choix de la valeur δ_t

Dans les premières expérimentations, nous avons par exemple noté que dans la plupart des cas, nous avions seulement 50% des systèmes qui étaient résolus. En outre, nous avons observé que sur certaines matrices, un solveur pouvait s'arrêter plus vite qu'un autre sans que le premier ne trouve une solution qui converge avec la tolérance que l'on s'est fixée. Dans ce cas la question

du meilleur solveur peut devenir difficile à définir car sur le temps, le premier solveur est le meilleur alors que sur la qualité de la solution, le second l'est. Nous avons traité ces cas ici en décidant qu'en cas de non convergence sur la tolérance fixée, le BICGSTAB est le meilleur. On pourrait néanmoins penser à une approche qui prend en compte les deux objectifs de minimiser le temps d'exécution ainsi que la norme des vecteurs résidus que l'on obtient dans la résolution des différents systèmes.

Les résultats que nous avons obtenus montrent que l'approche **ACPP** ne conduit pas à de bonnes prédictions sur des solveurs qui ne sont pas efficaces sur les instances du benchmark. Afin d'améliorer les tests que nous avons effectués, une piste intéressante consiste donc à choisir un benchmark dans lequel par exemple la méthode TFQMR sera moins dominée par les autres.

L'idée d'une phase d'exploration progressive s'inspire de l'exécution de **GAMBLETA**. Elle consiste à fixer une sous période Δ tel que pendant l'exploration à chaque période Δ on sélectionne un sous ensemble d'algorithmes selon le principe d'**ACPP** pour avec lesquels on on continue l'exploration. L'introduction de la période Δ est intéressante si très vite, l'on peut identifier à partir des profils d'exécution de l'instance qui est en cours de résolution quels sont les algorithmes qui n'auront pas sur celui ci un bon temps d'exécution. Toutefois, ce choix est pénalisant si l'on ne se trouve pas dans de telles situations.

En se basant sur les résultats obtenus dans les algorithmes *anytime*, on peut modifier l'algorithme précédent. Pour chaque algorithme candidat, étant données les exécutions obtenues du benchmark, on construit des modèles analytiques de sa qualité. Nous précisons qu'à un algorithme, on peut associer plus d'un modèle mathématique de qualité. Étant donnée une instance à résoudre, à partir de la phase d'exploration on collecte son profil effectif de performance et on essaie de déduire le modèle auquel il est le plus proche en employant par exemple la méthode des moindres carrés. L'intérêt d'utiliser un modèle analytique de la qualité est que potentiellement, on pourra réduire le temps des comparaisons nécessaires la détermination des plus proches voisins.

En dernière perspective, nous pensons qu'il faut introduire une stratégie efficace pour la détermination en pratique de δ_t. Sa valeur est en effet déterminante sur le surcoût induit par le changement de contexte.

Chapitre 4

Approche de sélection statique des portfolio d'algorithmes

Résumé : *Dans ce chapitre, nous présentons une approche statique de sélection des portfolio d'algorithmes. Dans cette approche, le problème majeur est celui du partage discret des ressources que nous définissons. La complexité de ce problème est étudiée et des algorithmes sont proposés pour sa résolution.*

4.1 Introduction

Dans le chapitre précédent, nous avons présenté une approche ayant pour objectif de déterminer étant donnée une liste d'algorithmes candidats résolvant un problème $\mathcal{P}$, la meilleure combinaison algorithmique pour résoudre chaque instance de $\mathcal{P}$. Dans la résolution de chaque instance I de $\mathcal{P}$, nous avons admis que l'on avait des attributs du problème calculable sur I et déterminants dans le comportement de I sur les différents algorithmes candidats. Comme attribut, nous avons utilisé le profil de performance.

L'hypothèse de la connaissance des informations déterminantes dans le comportement d'un algorithme sur une instance n'est toujours pas justifiée comme indiqué dans le chapitre 3. Sur des problèmes très difficiles et notamment les problèmes de la classe NP-complet, il existe très peu d'études théoriques permettant d'établir un lien entre des propriétés connues sur une instance et le comportement des algorithmes le résolvant. Si dans certaines situations, d'excellents résultats expérimentaux permettent de justifier de telles hypothèses [Xu et al. 2008, Gomes and Selman 2001], celles ci malgré tout restent limitées à des problèmes bien spécifiques. Face à cette difficulté, les approches dites on-line [Gagliolo and Schmidhuber 2008] [Borodin and El-Yaniv 1998] sont envisageables dans la mesure où elles ne tirent pas partie des informations issues du problème. Toutefois, les solutions qui y sont proposées admettent des hypothèses difficiles à vérifier comme la connaissance du coût induit par un mauvais choix

algorithmique étant donné une instance.

Dans ce chapitre, nous proposons une autre approche de combinaison des algorithmes dans laquelle nous ne prenons pas en compte la possibilité d'utiliser les informations extraites de l'instance en cours de résolution. Étant donnés un problème à résoudre et un ensemble d'algorithmes, notre cible est de trouver la combinaison d'algorithmes qui en moyenne conduit au plus temps d'exécution. En adoptant le modèle de combinaison par portfolio, nous proposons une formulation inspirée des travaux de [Sayag et al. 2006]. Ici, la meilleure combinaison algorithmique est représentée comme un portfolio définissant un partage de ressources entre les algorithmes. Une partie de l'approche que nous proposons ici a été publiée dans [Bougeret et al. 2009a, Bougeret et al. 2009b, Ngoko and Trystram 2009b]. En comparaison de l'approche proposée au chapitre 3, cette modélisation est statique dans la mesure où l'on cherche un seul portfolio d'algorithmes pour résoudre toutes les instances. Nous étudions ensuite la difficulté du problème de combinaison statique des algorithmes, nous proposons des algorithmes pour le résoudre et nous les validons expérimentalement à travers des simulations sur les algorithmes résolvant le problème de satisfiabilité (SAT) [Garey et al. 1979].

La suite du chapitre est organisée comme suit : Dans la Section 4.2, nous présentons quelques approches de combinaison statique d'algorithmes que nous analysons sommairement. Dans la Section 4.3 nous introduisons le problème de partage discret des ressources (*dRSSP*) que nous utiliserons pour modéliser la sélection statique des portfolio d'algorithmes. Dans la Section 4.4 nous proposons une approche exacte pour résoudre le *dRSSP*. La Section 4.5 est consacrée aux algorithmes d'approximation pour résoudre le *dRSSP*, la section 4.6 présente les résultats expérimentaux obtenus par sur les différents algorithmes que nous proposons et nous concluons à la Section 4.7.

4.2 Quelques formulations de la sélection statique des portfolio d'algorithmes

Dans cette partie, nous présentons deux modélisations du problème de sélection statique du meilleur portfolio d'algorithmes. Ce sont le problème de partage des ressources et le problème de partage de temps. Ces formulations ont été proposées dans [Sayag et al. 2006, Streeter et al. 2007].

4.2.1 Le problème de partage des ressources

Nous nous plaçons dans un contexte où pour un problème à résoudre, nous disposons d'une liste finie d'algorithmes candidats $\mathcal{A}$ ainsi qu'un benchmark $\mathcal{I}$ d'instances représentatives du problème. Étant donné que nous ne prenons pas en compte la possible utilisation des informations extraites de l'instance pour déterminer la meilleure combinaison d'algorithmes à appliquer, une approche consiste donc à trouver pour toutes les instances à résoudre une combinaison fixe d'algorithmes candidats.

Une telle approche a été suggérée dans [Sayag et al. 2006] avec le problème de partage de ressources (*RSSP*). L'hypothèse de base ici est proche de l'idée du voisinage d'instances introduite au chapitre 3. On admet que le comportement (en temps d'exécution) de n'importe quelle instance à résoudre sur les algorithmes candidats est similaire à celui d'une instance du benchmark. Ensuite, étant donnée une instance quelconque à résoudre, on ne connaît pas a priori l'instance du benchmark dont elle est proche. Cette hypothèse permet de ramener l'espace des instances du problème en l'espace fini des instances $\mathcal{I}$. Le meilleur portfolio d'algorithmes peut ainsi être recherché dans une configuration où l'on a des instances un jeu d'instances $\mathcal{J} \subseteq \mathcal{I}$ à résoudre sans que l'on ne sache en résolvant une instance $I \in \mathcal{J}$ à quelle instance de $\mathcal{I}$ elle correspond. En admettant les ressources et le temps continu sur le benchmark des instances $\mathcal{I}$, le *RSSP* peut être décrit comme suit :

Resource Sharing Scheduling Problem (RSSP)

Instance : Un ensemble fini d'instances $\mathcal{I} = \{I_1, \ldots, I_n\}$, un ensemble fini d'algorithmes $\mathcal{A} = \{A_1, \ldots, A_k\}$, des coûts $C(A_i, I_j) \in R^+$ pour chaque $I_j \in I$, $A_i \in H$, un nombre réel $T \in R^+$.

Question : Existe t'il un vecteur $S = (S_1, \ldots, S_k)$ avec $S_i \in [0, 1]$ et $\sum_{i=1}^{k} S_i \leq 1$ tel que $\sum_{j=1}^{n} \min_{1 \leq i \leq k} \left\{ \frac{C(A_i, I_j)}{S_i} | S_i > 0 \right\} \leq T$?

Dans cette formulation, on doit affecter une fraction S_i des ressources à chaque algorithme A_i. On suppose que pour chaque instance I_j et algorithme A_i la valeur $\frac{C(A_i, I_j)}{S_i}$ donne le temps d'exécution de I_j sur A_i lorsqu'il est exécuté avec une fraction de S_i ressources. Ceci signifie que la performance d'un algorithme est proportionnelle au nombre de ressources dont il dispose, ce qui est parfois observé sur des algorithmes parallèles. Étant donnée une instance à résoudre, on exécute tous les algorithmes A_i sur elle jusqu'à ce qu'un des algorithmes termine son exécution. Aussi, le temps d'exécution de cette instances est $\min \left\{ \frac{C(A_i, I_j)}{S_i} | S_i > 0 \right\}$.
La redondance des calculs nécessaires à la résolution d'une instance dans le *RSSP* peut être perçue comme un handicap. Néanmoins, nous remarquons que lorsque les différences de performance entre algorithmes sont importantes, cette redondance peut être infime en comparaison d'un mauvais choix d'algorithmes.

Dans le *RSSP*, l'hypothèse de la connaissance des informations extractibles sur les instances n'est pas prise en compte. L'un de ses principaux défauts du *RSSP* est le partage en fraction non entières des ressources. Considérons par exemple un contexte parallèle et distribué à m ressources. Ici, les ressources peuvent idéalement être prises comme étant les processeurs. Implémenter dans un tel contexte certains partages continus de ressources peut s'avérer difficile. Une approche pour cela peut consister à attribuer dans l'exécution un même processeur à plusieurs algorithmes. Seulement, il devient alors difficile d'avoir une fonction de coût précise dès lors que des exécutions concurrentes d'algorithmes sur un même processeur entraîneront des défauts de caches et des ordonnancements de calculs difficilement prévisibles.

Une autre formulation similaire au *RSSP* a été proposée dans [Sayag et al. 2006] puis

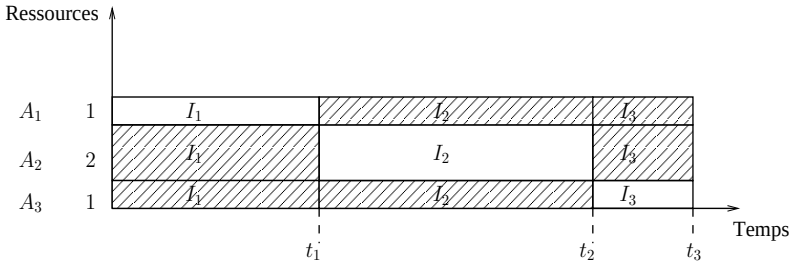


FIG. 4.1 – Exemple d'exécution concurrente de trois algorithmes sur trois instances avec un modèle de partage de ressources. Ici, $S_1 = \frac{1}{4}, S_2 = \frac{1}{2}$ et $S_3 = \frac{1}{4}$. On a ici trois instances résolues consécutivement. A la date t_1, I_1 est résolu par A_1 ; A la date t_2, I_2 est résolu par A_2 et à la date t_3, I_3 est résolu par A_3. Les zones hachurées représentent les portions de calcul qui ne seront pas utilisés dans la résolution des instances.

dans [Streeter et al. 2007]. Nous allons la présenter dans la suite.

4.2.2 Le problème de partage du temps

Le problème de partage de temps entre algorithmes (*TSSP*) exploite l'idée de l'exécution concurrente des algorithmes dans le temps. Dans cette formulation, on suppose que les algorithmes sont interruptibles et que le temps est discrétisé en unités de taille δ_t. Les algorithmes sont combinés en utilisant le modèle du portfolio par partage de temps. En admettant que l'exécution d'un algorithme sur une instance peut prendre au plus B unités de temps, un portfolio d'algorithmes par partage de temps se définit comme une fonction $\sigma : \{1, \ldots, kB\} \longrightarrow \mathcal{A}$. Avec une telle fonction, si l'on commence l'exécution d'une instance à la date t_0. Alors entre les dates $t_0 + (\alpha - 1)\delta_t$ et $t_0 + \alpha\delta_t$ l'on exécute l'algorithme $\sigma(\alpha)$.

Le coût de résolution d'une instance sur un portfolio d'algorithmes par partage de temps est déterminé comme suit : Étant donné un portfolio par partage de temps σ, soit $C(A_i, I_j)$ le temps nécessaire à la résolution de l'instance I_j sur l'algorithme A_i (on ne prend pas en compte le parallélisme) et $C(\sigma, I_j)$ le temps de résolution de l'instance I_j avec le portfolio σ. $C(\sigma, I_j)$ se définit comme la plus petite date t telle que pour un algorithme A_i, $C(A_i, I_j) = |\{t_1 < t : \sigma(t_1) = A_i\}|$. A partir de cette définition, le problème du partage de temps entre algorithmes peut être décrit comme suit :

Task Switching Scheduling Problem (TSSP)

Instance : Un ensemble fini d'instances $\mathcal{I} = \{I_1, \ldots, I_n\}$, un ensemble fini d'algorithmes $\mathcal{A} = \{A_1, \ldots, A_k\}$, des coûts $C(A_i, I_j) \in R^+$ pour chaque $I_j \in I$, $A_i \in H$, un nombre réel $T \in R^+$.

Question : Existe t'il une application $\sigma : \{1, \ldots, kB\} \longrightarrow \mathcal{A}$, telle que $\sum_{j=1}^{n} C(\sigma, I_j) \leq T$?

Le problème du partage de temps entre algorithmes est similaire au *RSSP*. On n'a pas ici le problème de partage en fraction des ressources mais on admet l'interruptibilité des algorithmes. Malgré tout, il reste toujours difficile d'avoir une fonction de coût précise. En effet, les interruptions et les redémarrages d'algorithmes entraîneront dans plusieurs cas des défauts de cache et des sauvegardes et des chargements de contexte qu'il est difficile d'évaluer. Aussi en pratique, étant donnée une instance I, l'estimation de la durée de sa résolution $C(\sigma, I)$ sur un portfolio σ peut être très mauvaise. Dans la définition du temps de résolution d'une instance dans le partage de temps, il faut donc introduire le surcoût nécessaire au changement de contexte.

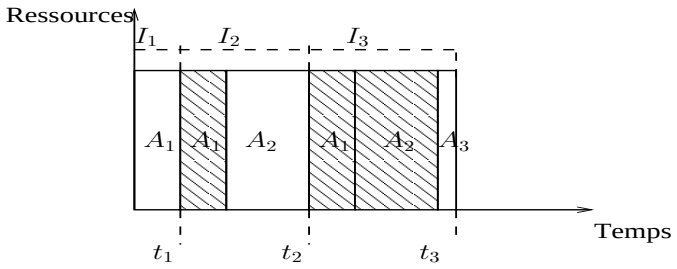


FIG. 4.2 – Exemple d'exécution concurrente de trois algorithmes sur trois instances avec un modèle de partage de temps. On exécute ici successivement A_1 puis A_2 et A_3. A la date t_1, I_1 est résolu par A_1 ; A la date t_2, I_2 est résolu par A_2 et à la date t_3, I_3 est résolu par A_3. La zone hachurée représente les exécutions qui ne permettent pas de résoudre en fin de compte l'instance en cours. Les zones hachurées représentent les calculs qui ne serviront pas en fin de compte dans la résolution d'une instance.

Les modèles proposés dans [Sayag et al. 2006, Streeter et al. 2007] sont intéressants pour définir une stratégie de combinaison d'algorithmes qui n'extraie aucune information sur l'instance à résoudre. Ils proposent toutefois des fonctions de coûts qui ne sont pas toujours faciles à obtenir en pratique. Dans le cas du *TSSP*, il faut en particulier introduire les surcoûts liés au changement de contexte. Ce surcoût est dépendant du mécanisme que l'on adopte pour la sauvegarde et la reprise du contexte d'exécution et donc difficile à estimer sans implantation algorithmique. Nous allons dans la suite considérer uniquement le modèle de partage de ressources (*RSSP*). Pour rendre le *RSSP* plus réaliste, il faut en particulier éviter des partages en fraction non entières de ressources. Nous allons dans la suite proposer une adaptation du *RSSP* et permettant de prendre plus en compte les spécificités de l'environnement d'exécution des algorithmes.

4.3 Le problème de partage discret des ressources

Dans cette section, nous proposons une version discrète du *RSSP* que nous notons *dRSSP*. Nous analysons ensuite sa complexité.

4.3.1 Définition

le *dRSSP* est une adaptation du *RSSP* dans laquelle on suppose que l'on est dans un contexte parallèle avec des ressources identiques et discrètes. Le coût d'exécution d'un algorithme est défini en prenant en compte le nombre de ressources sur lequel il est exécuté. Formellement, ce problème peut être défini comme suit :

discrete Resource Sharing Scheduling Problem (dRSSP)

Instance : Un ensemble fini d'instances $\mathcal{I} = \{I_1, \ldots, I_n\}$, un ensemble fini d'algorithmes $\mathcal{A} = \{A_1, \ldots, A_k\}$, un ensemble de m ressources identiques, des coûts $C(A_i, I_j, p) \in R^+$ pour chaque $I_j \in \mathcal{I}$, $A_i \in \mathcal{A}$ et $p \in \{1, \ldots, m\}$, un nombre réel positif T.

Question : Existe t'il un vecteur $S = (S_1, \ldots, S_k)$ avec $S_i \in \{0, \ldots, m\}$ and $0 < \sum_{i=1}^{k} S_i \leq m$ tel que $\sum_{j=1}^{n} \min_{1 \leq i \leq k} \{C(A_i, I_j, S_i) | S_i > 0\} \leq T$?

Le *dRSSP* suppose que les coûts d'exécution sont définis pour chaque algorithme, instances et nombre de ressources. On peut par ailleurs envisager une extension de définition en contexte hétérogène en définissant le coût d'exécution en fonction du sous ensemble des ressources considérées. Dans la suite, sauf mention contraire, nous posons $C(A_i, I_j) = C(A_i, I_j, m)$.

Dans un contexte parallèle, il est logique de supposer que les coûts d'exécution décroissent lorsque la quantité de ressources affectées aux algorithmes est importante. Dans la suite, nous allons continuer avec une version restreinte du *dRSSP* (le *l-dRSSP*) dans laquelle nous ajoutons une hypothèse de linéarité. Cette hypothèse indique que les coût d'exécutions exécution sont proportionnels au nombre de ressources utilisées ($C(A_i, I_j, p) = \frac{C(A_i, I_j, m)m}{p}$). Cette hypothèse est en particulier justifiée sur les algorithmes parallèles qui sont efficaces.

Afin de cerner la différence entre le *l-dRSSP* et le *RSSP* nous considérons l'exemple suivant.

Exemple 4.3.1. *On considère un partage de ressources à définir sur* 2 *ressources avec* 2 *algorithmes(*A_1*,* A_2*) et* 2 *instances (*I_1, I_2*). On suppose que la matrice des coûts sur l'ensemble des ressources est la suivante :*

C(.)	I_1	I_2
A_1	2	10
A_2	10	1

TAB. 4.1 – Matrice des coûts d'exécution

Pour déduire la solution optimale avec le *RSSP*, une fraction x d'une ressource doit être allouée de sorte à minimiser : $\min\left(\frac{2}{x}, \frac{10}{1-x}\right) + \min\left(\frac{10}{x}, \frac{1}{1-x}\right)$. Le minimum est obtenu pour $x = 2 - \sqrt{2}$. Ainsi, la solution optimale avec le *RSSP* consiste à donner $2(2 - \sqrt{2})$ ressources à A_1 et $2(\sqrt{2} - 1)$ ressources à A_2. Cette solution nous conduit à un coût total de résolution des instances de 5.8284. La solution optimale avec le *l-dRSSP* consiste à donner une ressource à A_1

et une autre à A_2. Cette solution conduit à un coût total de résolution de 6. Ce résultat peut être généralisé comme suit :

Propriéte 4.3.1. *Le temps moyen de résolution des instances $\mathcal{I}$ avec le* l-dRSSP *est supérieur ou égal à celui pris pour le* RSSP

La propriété 4.3.1 est facile à déduire car toute solution valide pour le *l-dRSSP* l'est aussi pour le *RSSP* sans que la réciproque ne soit vraie. Dans la suite, nous étudions la complexité du *l-dRSSP*.

4.3.2 Complexité

Théorème 4.3.1. l-dRSSP *est NP-Complet.*

Démonstration. Il est facile de vérifier que ce problème est dans NP. Supposons qu'un vecteur $S = (S_1, \ldots, S_k)$ soit une solution candidate à une instance du *l-dRSSP* avec m ressources et k algorithmes. On peut vérifier en $O(nk)$ que S vérifie toutes les conditions d'une solution au problème en procédant comme suit : pour chaque instance I_j on trouve la valeur $v_j = \min\limits_{1\leq i\leq k}\{\frac{C(A_i,I_j)}{S_i}|S_i > 0\}$. Ceci peut être fait en $O(k)$. Pour les n instances on peut déterminer la somme des $v_j, j = 1, \ldots, n$ en $O(nk)$.

Nous allons procéder par une réduction au problème de couverture d'ensembles (unweighted set cover [Hochbaum 1997]) pour montrer que le *l-dRSSP* est NP complet.

Le problème de couverture d'ensembles est le suivant : Étant donné une valeur m et les ensembles finis $U = \{u_1, \ldots, u_n\}$, $F = \{F_1, \ldots, F_k\}$ avec $\bigcup_{i=1}^{k} F_i = U$, trouver un sous ensemble de $F^c \subseteq F$ tel que $\bigcup_{F_i\in F^c} = U$ et $|F^c| \leq m$. Pour le résoudre, nous allons utiliser le *l-dRSSP* comme suit :
On construit une instance du *l-dRSSP* avec m ressources à partager où $\mathcal{I} = \{I_1, \ldots, I_n\}$, tel que chaque $I_j \in \mathcal{I}$ correspond à $u_j \in U$, $\mathcal{A} = \{A_1, \ldots, A_k\}$ avec A_i qui correspond à F_i. On fixe

$$C(A_i, I_j, m) = \begin{cases} \alpha > 0 & \text{si } u_j \in F_i \\ nm\alpha + 1 & \text{sinon.} \end{cases}$$

et $T = nm\alpha$. On résout ce problème et si l'on obtient une solution, on prend l'ensemble $F^c = \{F_i \text{ t.q. } S_i \neq 0\}$ comme solution au problème de couverture d'ensembles. Le cas échéant, on répond que la couverture est impossible

La réduction proposée est bien polynomiale. Par ailleurs, l'ensemble F^c est de taille au plus m étant donné que nous avons m ressources. Cet ensemble est aussi une couverture d'ensembles car s'il existait un élément u_j tel qu'aucun ensemble F^c ne le contienne, le coût du problème de *l-dRSSP* associé serait au moins $nm\alpha + 1$ à cause du temps d'exécution de I_j dans la solution obtenue.

Réciproquement, s'il existe une couverture d'ensemble F^c de taille au plus m, alors le vecteur $S = (S_1, \ldots, S_k)$ où $\forall i, 1 \leq i \leq k\ S_i = \begin{cases} 1 & \text{si } F_i \in F^c \\ 0 & \text{sinon.} \end{cases}$ est une solution au problème du *l-dRSSP* décrit plus haut. En effet, chaque instance I_j aurait un algorithme avec un coût d'exécution α associé à qui une ressource serait associée, cela conduit à un coût d'exécution total de $nm\alpha$. ❑

Corollaire 4.3.1. dRSSP *est NP complet*

Les résultats ci dessus montrent qu'à moins que $P = NP$, on ne peut pas trouver d'algorithmes qui en temps polynomial donne une solution au *l-dRSSP*.

Dans la démonstration précédente, l'on peut noter que la NP complétude est obtenue parce que nous avons éventuellement plus d'algorithmes candidats que de ressources ($k > m$). Si nous avons en effet plus de ressources que d'algorithmes, le problème de couverture d'ensembles n'est pas difficile. En pratique dans un contexte parallèle et distribué, l'on peut supposer que l'on aura en général plus de ressources que d'algorithmes. Il est donc intéressant de savoir si dans ce cas le *l-dRSSP* reste difficile.

Théorème 4.3.2. l-dRSSP *est NP-Complet même lorsque l'on a plus de ressources que d'algorithmes.*

Démonstration. Nous allons exploiter la réduction précédente ici. L'idée ici est d'introduire une instance et un algorithme virtuels de sorte que l'on ne puisse atteindre la borne fixée sur le *l-dRSSP* que si l'on ne donne suffisamment des ressources à l'algorithme virtuel afin d'obtenir un *l-dRSSP* à résoudre avec plus d'algorithmes que de ressources sur les instances et ressources restantes.

On considère un problème de couverture d'ensembles donné par : une valeur $m \geq 1$ et les ensembles finis $U = \{u_1, \ldots, u_n\}$, $F = \{F_1, \ldots, F_k\}$ avec $\bigcup_{i=1}^{k} F_i = U$, tel que l'on cherche un sous ensemble de $F^c \subseteq F$ avec $\bigcup_{F_i \in F^c} = U$ et $|F^c| \leq m$. Nous procédons comme suit pour le résoudre :
Nous construisons une instance du $l-dRSSP$ avec m ressources à partager où $\mathcal{I} = \{I_1, \ldots, I_{n+1}\}$, tel que chaque $I_j, j \leq n$ correspond à $u_j \in U$, $\mathcal{A} = \{A_1, \ldots, A_{k+1}\}$ avec $A_i, i \leq k$ qui correspond à F_i. On fixe

$$C(A_i, I_j, (k+1)m) = \begin{cases} \alpha > 0 & \text{si } (i \leq k, j \leq n \text{ et } u_j \in F_i \\ k\gamma & \text{si}(i = k+1 \text{ et } j = n+1) \\ T+1 & \text{sinon.} \end{cases}$$

Ici, $T = \gamma(k+1) + nm(k+1)\alpha$ et $\gamma = nm\alpha(km-1)+1$.

On résout ce problème en supposant que l'on a $(k+1)m$ ressources et si l'on obtient une solution, on prend l'ensemble $F^c = \{F_i \text{ t.q. } S_i \neq 0\}$ comme solution au problème de couverture d'ensembles. Le cas échéant, on répond que la couverture est impossible.

On peut noter que dans cette réduction, le nombre d'algorithmes candidats $(k+1)$ est plus petit que le nombre de ressources $(k+1)m$. Nous avons choisi γ tel que si l'on affecte moins de km ressources à A_{k+1}, alors il ne serait pas possible d'avoir une solution au *l-dRSSP* ayant un coût inférieur ou égal à T car alors, le coût d'exécution de I_{n+1} sur A_{k+1} serait plus grand que T [1]. Aussi, toute solution au problème que nous avons défini donne une solution au *l-dRSSP* avec les instances $I_1, \ldots, I_n$ et algorithmes $A_1, \ldots, A_k$ sur au plus m ressources ; Or, ceci correspond à une solution au problème de couverture d'ensembles.

Nous avons choisi ici $(k+1)m$ ressources juste pour illustrer un nombre de ressources supérieur à un nombre d'algorithmes. On peut obtenir toutefois une preuve similaire en utilisant tout nombre de ressources plus grand que k. ❑

La NP complétude du *l-dRSSP* est établie même lorsque l'on a plus de ressources que d'algorithmes. Dans la suite, nous allons proposer des algorithmes permettant de résoudre le *l-dRSSP*. Nous allons nous intéresser à des approches exactes et heuristiques avec garanties. Nous commençons par les approches exactes

4.4 Résolution exacte du *l-dRSSP*

Une solution au *l-dRSSP* est un vecteur $S = (S_1, \ldots, S_k)$ avec $\sum_{i=1}^{k} S_i \leq m$ où m est le nombre de ressources et k le nombre d'algorithmes. Dès lors, une approche naive de résolution du *l-dRSSP* consiste à considérer toutes les allocations S^q avec $0 \leq S_i^q \leq m$. Parmi ces allocations, on sélectionne le sous ensemble dans lequel chaque allocation q vérifie $\sum_{i=1}^{k} S_i^q \leq m$. Pour ce sous ensemble, on calcule le coût moyen de résolution des instances. L'allocation conduisant au plus petit coût moyen de résolution est alors choisie comme solution. Il est facile de vérifier que cette approche donne une solution exacte au *l-dRSSP* après un parcours de $(m+1)^k$ allocations. Toutefois, l'on peut remarquer que l'on génère ici des allocations invalides (celles pour lesquelles on a plus de m ressources affectées aux processeurs) que l'on élimine par la suite. Dans la suite, nous allons proposer un algorithme qui ne génère aucune allocation invalide.

4.4.1 Algorithme

Considérons un vecteur $S = (S_1, \ldots, S_k)$. Nous procédons ici en donnant différentes valeurs aux S_i. Afin de ne générer que des allocations valides, nous allons générer chaque allocation S en affectant successivement des valeurs (correspondant à des ressources) à $S_1, S_2, \ldots$. jusqu'à ce que l'on n'ait plus de ressources à allouer (auquel cas le reste des allocations est défini comme étant nul ou que l'on n'ait atteint S_k). Lorsque nous allouons une ressource à

[1] Les détails sur le choix de γ et T sont donnés en annexe pour une meilleure présentation de la preuve.

S_i, nous prenons en compte la quantité de ressources déjà allouée à $S_1, \ldots, S_{i-1}$. Nous utilisons pour cela deux variables $index$ et sum. La variable $index$ nous indique l'allocation S_{index} que nous voulons définir alors que la variable sum nous donne le nombre de ressources que nous avons déjà utilisés. Nous utilisons aussi une variable $melCout$ qui nous donne le meilleur coût obtenu étant donné une allocation. Le principe ci dessus est implémenté dans l'algorithme **ParcoursdRSSP** que nous présentons ci dessous.

Algorithm 1 ParcoursdRSSP(S, $index$, sum, $melCout$, Sol)

1: **if** $index = k$ **then**
2: $S_k = m - sum$ {Pour le dernier algorithme, on donne le nombre de ressources qui reste}
3: $Cout = Cout\_Portfolio(S)$ {On calcule le coût du portfolio induit par l'allocation S et on sauvegarde cela dans $Cout$}.
4: **if** $Cout < melCout$ **then**
5: $melCout = Cout$
6: **for** $j = 1$ to k **do**
7: $Sol_i = S_i$
8: **end for**
9: **end if**
10: **else**
11: **if** $sum < m$ **then**
12: **for** $j = 0$ to $m - sum$ **do**
13: $S_{index} = j$ {Si toutes les ressources ne sont pas allouées ($sum < m$), on teste toutes les allocations possibles pour l'algorithme $index$ puis on passe aux allocations possibles pour $index + 1$}
14: **ParcoursdRSSP**(S, $index + 1$, $sum + j$, $melCout$, Sol)
15: **end for**
16: **else** { Toutes les ressources sont allouées, on avance $index$ pour calculer le coût de l'allocation}
17: **for** $j = index$ to $k - 1$ **do**
18: $S_j = 0$
19: **end for**
20: **ParcoursdRSSP**(S, k, sum, $melCout$, Sol)
21: **end if**
22: **end if**

Dans **ParcoursdRSSP**, on procède récursivement pour parcourir toutes les allocations possibles de ressources. A la fin du parcours, la variable Sol contient l'allocation qui conduit au plus petit temps d'exécution. L'algorithme doit être lancé initialement avec les valeurs de $index, sum$ qui sont nulles et $melCout = +\infty$.

On peut facilement établir que **ParcoursdRSSP**, ne génère que des allocations valides. Pour un k fixé, **ParcoursdRSSP** donne une solution en temps pseudo polynomial au *l-dRSSP*. Cette solution toutefois sera coûteuse en temps si k est très grand. Dans la suite, nous allons nous intéresser à la construction des heuristiques efficaces avec un compromis raisonnable entre le

temps d'exécution et la qualité de la solution obtenue sur le *l-dRSSP*.

4.5 Heuristiques de résolution du *l-dRSSP*

Dans cette partie, nous nous intéressons à la construction d'heuristiques pour le *l-dRSSP*. Nous ciblons en particulier les heuristiques garanties. Nous commençons par présenter un résultat sur l'inapproximabilité générale du *l-dRSSP*. Nous définissons ensuite une borne inférieure pour analyser les heuristiques sur le *l-dRSSP* puis nous proposons dans une version restreinte du problème plusieurs heuristiques.

4.5.1 Inapproximabilité du *l-dRSSP*

Proposition 4.5.1. *Le* l-dRSSP *ne peut pas être approximé en facteur constant et temps polynomial si* $P \neq NP$.

Démonstration. Ce résultat peut facilement être obtenu par *gap reduction* en exploitant la preuve de NP complétude [Garey et al. 1979]. Dans cette réduction, l'on substitue :

$$C(A_i, I_j, m) = \begin{cases} \alpha > 0 & \text{si } u_j \in F_i \\ nm\alpha + 1 & \text{sinon.} \end{cases}$$
$$\text{et } T = nm\alpha.$$

par

$$C(A_i, I_j, m) = \begin{cases} \alpha > 0 & \text{si } u_j \in F_i \\ \lambda(nm\alpha) + 1 & \text{sinon.} \end{cases}$$
$$\text{et } T = nm\alpha.$$

Si l'on obtient une solution au *l-dRSSP* de coût $\leq \lambda T$ alors il est certain que le sous ensemble d'algorithmes ayant au moins une ressource garantit que pour toute instance $I_j, \exists A_i | S_i \geq 1$ et $C(A_i, I_j, m) = \alpha$. Une telle solution donne donc une couverture d'ensembles. ❑

Ce résultat montre que dans le cas général, l'on ne peut pas approximer en facteur constant le *l-dRSSP*. Toutefois, il faut noter qu'il est inspiré de la preuve de NP complétude qui considère en particulier le *l-dRSSP* quand on a plus d'algorithmes que de ressources. On peut donc espérer que quand on a plus de ressources que d'algorithmes ($k < m$), il est possible de trouver un algorithme d'approximation au *l-dRSSP*. En outre, cette hypothèse est raisonnable en pratique. Nous allons dans la suite considérer uniquement ce cas.

4.5.2 Bornes inférieures

Afin de pouvoir approximer le *l-dRSSP* dans le cas restreint, nous allons utiliser une borne inférieure sur le coût total que l'on peut espérer dans une solution. Une approche envisageable pour une borne inférieure consiste à prendre la solution optimale du *RSSP* dans le cas $k < m$. Ce résultat peut facilement déduit de la propriété 4.3.1. Le choix d'une telle borne peut aussi être motivé par la construction des algorithmes d'approximation pour les problèmes NP complet à partir de la relaxation d'un programme linéaire en nombre entiers [Hochbaum 1997]. On considère ici le *RSSP* comme étant la version *relaxée* du *l-dRSSP*.
Considérer le *RSSP* comme borne inférieure n'est toutefois pas adéquat. En effet, il est difficile de résoudre le *RSSP* par programmation linéaire car dans [Sayag et al. 2006], les auteurs montrent que ce problème est NP complet. La meilleure solution connue pour le *RSSP* étant en $O(n^k)$, on aurait un algorithme en temps pseudo polynomial au plus en arrondissant cette solution. On peut toujours penser construire une solution au *l-dRSSP* à partir d'une approximation du *RSSP* avec $k < m$. Dans ce dernier cas, nous avons le résultat suivant :

Proposition 4.5.2. *Soit C_1 le coût de traitement des instances avec le* l-RSSP *et C_2 celui obtenu avec le* dRSSP. *On peut avoir un jeu d'instances tel que $\frac{C_1}{C_2} \leq \frac{m}{m-k+1} + o$ avec $o \ll 1$*

Démonstration. Ce résultat est obtenu dans le cas où l'on a n instances avec $n > k$. Et les coûts sont tels que :

$$C(A_i, I_j, m) = \begin{cases} \epsilon & \text{si } i = j \text{ et } j < k\ , \\ \gamma & \text{si } i = 1 \text{ et } j \geq k \\ B & \text{sinon.} \end{cases}$$

avec B qui est une très grande valeur et $\epsilon \ll \gamma \ll nm\gamma \ll B$. En choisissant n et B suffisamment grand, l'on peut avoir une configuration où la solution donnée par *RSSP* donne $m - x$ ressources à A_1 et partage les $x < 1$ ressources restantes aux $k - 1$ algorithmes restants. La solution par contre du *l-dRSSP* donne au moins une ressource à tout le monde car la valeur de B est très grande et $m - k - 1$ ressources à A_1. Avec ces données, le coût total donné par la solution du *l-dRSSP* est $C_1 = ((n-k+1)\gamma\frac{m}{m-k-1} + m(k-1)\epsilon$. Le coût du *RSSP* en négligeant le coût de traitement des instances $I_1, \ldots, I_{k-1}$ (car ϵ est très petit) est $C_2 \leq (n-k+1)\gamma\frac{m}{m-x}$.

$$\frac{C_1}{C_2} \leq \frac{m-x}{m-k+1} + \frac{(m-x)(k-1)\epsilon}{\gamma(n-k+1)}.$$
$$\backsim \frac{m-1}{m-k+1}$$

❑

En prenant $m = k$, ce résultat suggère qu'à partir d'une α approximation au *RSSP*, il sera difficile de construire une approximation meilleure que $k\alpha$ du *l-dRSSP* en prenant comme borne inférieure la solution optimale au *RSSP*. Dans la suite, nous allons utiliser une borne inférieure

plus simple à manipuler et qui nous permet d'avoir des approximations tout aussi bonnes qu'en se basant sur le *RSSP*.

En supposant que p_{max} est le plus grand nombre de processeurs affecté à un algorithme dans la solution optimale, nous proposons alors d'utiliser comme borne inférieure, la valeur $Lb = \frac{m}{p_{max}} \sum_{j=1}^{n} \min_i \{C(A_i, I_j, m)\}$.
L'idée dans cette borne inférieure est d'approcher le temps de résolution de chaque instance par le plus petit temps possible que l'on peut obtenir dans la solution optimale. On peut facilement éliminer p_{max} dans cette formule car $p_{max} \leq m$. Nous allons toutefois garder cela pour donner une idée plus précise des approximations que nous obtiendrons.
La borne Lb est facilement atteignable dans l'exemple suivant :

Exemple 4.5.1. *On considère une instance* l-dRSSP *contenant* k *instances* $I_1, \ldots, I_k$ *dans laquelle on a les coûts de résolution :*

$$C(A_i, I_j, m) = \begin{cases} \epsilon & \textit{si } j = i \\ B & \textit{sinon.} \end{cases}$$

B *ici est tel que* $mk\epsilon < B$.

Avec les données de l'exemple 4.5.1, il faut donner une ressource à tous les algorithmes dans la solution optimale ce qui correspond à la borne inférieure.
Dans la suite, nous verrons comment construire des algorithmes d'approximation avec Lb comme borne inférieure. Nous adopterons les notations suivantes :

- Pour chaque instance I_j, $C^*(I_j) = \min_i \{C(A_i, I_j, m)\}$.
- $\mathcal{A}^* = \{A_i \in \mathcal{A}$ ssi $\exists j \ | C^*(I_j) = \min_i \{C(A_i, I_j, m)\}$.

4.5.3 Algorithmes par classification-optimisation

Une des difficultés dans la résolution du *l-dRSSP* vient du fait que l'on ne connaît pas dans la solution optimale étant donnée une instance quelconque, l'algorithme candidat qui sur celui ci aura le plus petit coût de résolution. Si l'on connaissait cela pour toute instance, il serait beaucoup plus simple de trouver la répartition des ressources à opérer. Suivant cette logique la résolution du *l-dRSSP* peut être abordée par une approche dite de classification-optimisation qui procède en parcourant des affectations possibles d'instances aux algorithmes pour lesquelles ont détermine ensuite la répartition des ressources correspondantes. Plusieurs approches ont été proposés suivant cette logique.

Dans [Sayag et al. 2006], une solution exacte basée sur le parcours de toutes les affectations instances-algorithmes possibles a été proposée pour le *RSSP* . Cette solution est néanmoins exponentielle. Une autre solution cette fois heuristique que l'on pourrait appliquer dans notre

cas a été proposée dans [Petrik and Zilberstein 2006]. Cette solution est basée sur un modèle prédicteur-correcteur pour parcourir les différentes allocations possibles. Initialement, on considère une affectation d'instances aux algorithmes. Celle-ci est censée indiquer pour chaque instance quel est l'algorithme qui le résout en le plus petit temps d'exécution. Suivant cette affectation, on déduit la répartition des ressources permettant de minimiser le temps d'exécution total des instances. On obtient alors une répartition des ressources. Dans celle-ci, on vérifie si le plus petit temps de résolution des instances est donnée suit la prédiction que l'on s'est faite initialement. Si tel n'est pas le cas, on reconsidère la nouvelle prédiction des instances et on itère à nouveau. Sinon on s'arrête. En fonction de l'affectation initiale que l'on a faite, ce procédé peut ou pas converger. Néanmoins, même lorsqu'il converge, il n'est pas certain que l'on ne tombera pas sur un minimum local.

Les approches par classification-optimisation sont limitées par le nombre d'affectations instances-algorithmes que l'on peut avoir ($\leq k^n$). Face à cette difficulté on pourrait juste se réduire à une liste d'affectations que l'on jugerait pertinente. En particulier, l'affectation qui donne à tout algorithme les instances sur lesquelles il est le meilleur est intéressante. Ceci nous permet de définir l'algorithme de l'allocation proportionnelle qui procède comme suit :

PA (Proportional Allocation)

1. Étant donné un benchmark d'instances $\mathcal{I}$ et un ensemble d'algorithmes $\mathcal{A}$, on détermine pour chaque algorithme A_i le sous ensemble $\sigma_i \subseteq \mathcal{I}$ des instances sur lesquelles A_i a le plus petit temps de résolution.
2. On construit le sous ensemble d'algorithmes $\mathcal{A}^* \subseteq \mathcal{A}$ tel que $A_i \in \mathcal{A}^*$ si $\sigma_i \neq \emptyset$
3. A chaque algorithme candidat $A_i \in \mathcal{A}^*$, on affecte $S_i = \lfloor \frac{m.w_i}{\sum_{i=1}^{k} w_i} \rfloor$ ressources où $w_i = \sum_{I_j \in \sigma_i} C(A_i, I_j, m)$
4. On partage équitablement les $m - \sum_{i=1}^{k} \lfloor \frac{m.w_i}{\sum_{i=1}^{k} w_i} \rfloor$ ressources restantes aux algorithmes $\mathcal{A}^*$ en donnant la priorité à ceux ayant déjà le moins de ressources affectées.

On peut facilement établir que cet algorithme est en $O(nk)$. L'allocation proportionnelle attribue les ressources aux algorithmes proportionnellement au temps d'exécution des instances sur lesquels ceux ci sont les meilleurs. Dans le pire des cas, on peut facilement établir qu'elle va produire une solution m-approchée de la solution optimale. Cette situation est observable lorsque dans l'allocation proportionnelle on a un algorithme A_i ayant un poids w_i très faible en comparaison de celui des autres (de sorte que dans **PA** on ne lui affecte qu'une ressource) mais qui dans la solution optimale a toutes les ressources. Ceci arrive en particulier si le temps d'exécution du dit algorithme sur chaque instance est à chaque fois légèrement supérieur au temps d'exécution du meilleur algorithme. Ce résultat révèle une faiblesse de l'allocation proportionnelle. Celle-ci est d'autant plus importante que l'on peut établir que n'importe quelle allocation d'au moins une ressources aux algorithmes $\mathcal{A}^*$ donne une solution m-approchée pour *l-dRSSP*. Dans la suite, nous allons considérer une autre forme d'allocation de ressources aux algorithmes.

4.5.4 Algorithmes de l'allocation moyenne

La non connaissance de la bonne affectation instances-algorithmes rend difficile la construction d'une solution efficace pour *l-dRSSP*. Étant donnée une répartition des ressources quelconque, on peut toutefois faire l'observation suivante :

Propriéte 4.5.1. *Étant donné une solution au* l-dRSSP *de coût total* C. *On a* $\frac{C}{Lb} \leq \frac{p_{max}}{q_{min}}$ *où* q_{min} *est le plus petit nombre de ressources alloué à un algorithme* $A_i \in \mathcal{A}^*$ *dans la solution de coût* C.

Démonstration. Étant donné une instance I_j dans la solution son temps d'exécution est $C(I_j) \leq \frac{m}{q_{min}} C^*(I_j)$. Aussi,

$$\frac{C}{Lb} \leq \frac{(\frac{m}{q_{min}}) \sum_{j=1}^{n} C^*(I_j)}{(\frac{m}{p_{max}}) \sum_{j=1}^{n} C^*(I_j)} \leq \frac{p_{max}}{q_{min}}$$

❑

Ce résultat nous indique donc que le plus petit nombre de ressources alloués aux algorithmes dans $\mathcal{A}^*$ est déterminant pour avoir une bonne approximation au *l-dRSSP* avec la borne inférieure Lb. Ceci nous amène à proposer une stratégie dans laquelle les différents algorithmes ont pratiquement le même nombre de ressources. Cela est fait dans l'algorithme de l'allocation moyenne dit *Mean Allocation* (**MA**). Nous supposons ici que $u = |\mathcal{A}^*|$ et $A_1, \ldots, A_u \in \mathcal{A}^*$

Algorithm 2 MA(S)

```
1: q = m div k
2: r = m − k ∗ q
3: for i = 1 to u do
4:       S_i = q
5: end for
6: for i = 1 to r do
7:       S_i = S_i + 1
8: end for
```

Proposition 4.5.3. ***MA*** *est une* $(2u-1).\frac{p_{max}}{m}$ *approximation pour* l-dRSSP *en* $O(k)$

Démonstration. Nous allouons dans **MA** au moins $q, (m = qu + r)$ ressources à chaque algorithme (et donc aussi à celui qui résout I sur toutes les ressources avec le coût $C^*(I)$). Aussi, dans **MA** chaque instance I est résolue en un temps de $C(I) \leq \frac{m}{q} C^*(I) \leq (u + \frac{r}{q}) C^*(I)$. Étant donné que r est au plus égal à $u - 1$ et q est au moins égal à 1, nous avons : $C(I) \leq (2u-1) C^*(I)$. Aussi

$$\frac{\sum_{j=1}^{n} C(I_j)}{Lb} \leq \frac{(2u-1) \sum_{j=1}^{n} C^*(I_j)}{(\frac{m}{p_{max}}) \sum_{j=1}^{n} C^*(I_j)} \leq (2u-1).\frac{p_{max}}{m}$$

❑

Ce résultat montre que la qualité de **MA** dépend de la cardinalité de $\mathcal{A}^*$.

L'allocation moyenne est une solution garantie pour résoudre *l-dRSSP*. La cardinalité du nombre d'algorithmes à qui l'on affecte des ressources ($\mathcal{A}^*$) détermine sa qualité. Lorsque dans le jeu d'instances, on a plusieurs algorithmes candidats ayant une instance sur laquelle, ils ont le plus petit temps de résolution, l'allocation moyenne peut se révéler coûteuse inutilement. En particulier, considérons l'exemple qui suit :

Exemple 4.5.2. *Nous avons trois algorithmes candidats à déployer sur* 6 *ressources. La matrice des coûts de résolution de ces algorithmes est donnée ci dessous :*

C(.)	I_1	I_2	I_3
A_1	100	100	4
A_2	2	3	100
A_3	4	1	100

TAB. 4.2 – Coût d'exécution

En appliquant le procédé de sélection de l'allocation moyenne ici, on est obligé d'affecter deux ressources à tous les algorithmes. Ceci nous conduit à un coût total de résolution des instances de 21. Néanmoins, si l'on avait sélectionné uniquement deux algorithmes (par exemple A_1 et A_2), on aurait eu un coût total de 18. Cet exemple nous amène à modifier le procédé de sélection dans l'allocation moyenne pour prendre en compte de telles situations. Ceci est d'autant plus important que le nombre important d'algorithmes candidats devait plutôt contribuer à améliorer la qualité du portfolio d'algorithmes.

4.5.5 Réduction du nombre d'algorithmes dans l'allocation moyenne

Dans l'algorithme **MA**, nous avons sélectionné un sous ensemble d'algorithmes auxquels nous affectons des ressources. Avec cette sélection, nous éliminons des affections de ressources qu'il est difficile de prendre en compte dans la borne supérieure de **MA**. La sélection des algorithmes auxquels l'on affecte des ressources peut être généralisée en fixant un seuil Δ et en redéfinissant $\mathcal{A}^*$ comme étant le plus petit sous ensemble d'algorithmes $\mathcal{A}^\Delta \subseteq \mathcal{A}^*$ tel que si un algorithme $A_i \in \mathcal{A}^\Delta$ alors il existe une instance I_j tel que $C(A_i, I_j, m) = C^*(I_j) + \Delta$. Une approche similaire pour la réduction du nombre d'algorithmes dans l'allocation moyenne a été publiée dans [Ngoko and Trystram 2009b]. Nous avons le résultat suivant sur la détermination de l'ensemble $\mathcal{A}^\Delta$:

Proposition 4.5.4. *Le problème de trouver le plus petit sous ensemble* $\mathcal{A}^\Delta$ *est équivalent au problème de calcul de la plus petite couverture d'ensembles*

Démonstration. On considère la formulation suivante pour le problème de calcul de la plus petite couverture d'ensembles : Étant donnés les ensembles finis $U = \{u_1, \ldots, u_n\}$ et F où

$F = \{F_1, \ldots, F_k\}$ et $\bigcup_{i=1}^{k} F_i = U$ trouver un sous ensemble $F^c \subseteq F$ tel que $\bigcup_{F_i \in F^c} = U$ et $|F^c|$ est minimal. Dans le problème de calcul du plus petit sous ensemble $\mathcal{A}^{\Delta}$, il suffit de prendre $\mathcal{I} = U$, chaque algorithme A_i comme un ensemble F_i tel que $I_j \in F_i$ si A_i peut résoudre I_j en un temps au plus égal à $C^*(I_j) + \Delta$. ❑

Corollaire 4.5.1. *On peut calculer en temps polynomial une* $0.72.ln(n)$ *approximation* $\mathcal{A}^{\Delta}$ *du sous ensemble optimal.*

Ce résultat est évident en utilisant l'algorithme glouton du problème de calcul de la plus petite couverture d'ensembles [Feige 1998].
En utilisant le sous ensemble $\mathcal{A}^{\Delta}$, nous pouvons lui partager les ressources comme dans l'algorithme **MA**. Nous obtenons alors l'algorithme **MA(Δ)**. L'intérêt dans cet algorithme est que si la valeur de Δ est bien choisie, la cardinalité de $\mathcal{A}^{\Delta}$ peut être plus petite que celle de $\mathcal{A}^*$.

Proposition 4.5.5. *Soit* $u_{\Delta} = \mathcal{A}^{\Delta}$, ***MA(Δ)*** *est une* $(2u_{\Delta} - 1).\frac{p_{max}}{m}(1 + n\epsilon)$ *approximation pour* l-dRSSP *en temps polynomial avec* $\epsilon = \frac{\Delta}{Lb}$.

Démonstration. La preuve est similaire à celle de **MA**. Pour chaque instance I_j étant donné que chaque algorithme dans $\mathcal{A}^{\Delta}$ a au moins q ressources $(m = qu_{\Delta} + r)$ alors on a $C(I_j) \leq (2u_{\Delta} - 1)(C^*(I_j) + \Delta)$. Aussi,

$$\begin{aligned}\frac{\sum_{j=1}^{n} C(I_j)}{Lb} &\leq (2u_{\Delta} - 1).\frac{p_{max}}{m}.\frac{\sum_{j=1}^{n} C^*(I_j) + n\Delta}{\sum_{j=1}^{n} C^*(I_j)}. \\ &\leq (2u_{\Delta} - 1).\frac{p_{max}}{m}(1 + n\epsilon)\end{aligned}$$

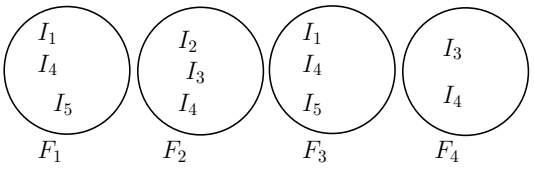
❑

Ce résultat montre que si $u_{\Delta} < u$, **MA(Δ)** peut avoir une meilleure approximation que **MA**.

Nous avons proposé un ensemble d'algorithmes qui permettent de réduire la taille du sous ensemble sur lequel on partage des ressources dans l'algorithme **MA**. Dans ces algorithmes un seuil Δ doit être donné en entrée pour définir la sélection des algorithmes. La difficulté de cette approche consiste à choisir judicieusement la valeur de Δ de sorte à garantir une qualité de solution au moins aussi bonne que celle de **MA**. Le résultat suivant donne une borne sur le choix optimal de cette valeur.

Proposition 4.5.6. *La valeur de* Δ *conduisant au plus petit coût d'exécution en moyenne dans* ***MA(Δ)*** *peut être choisie en parcourant* $O(nk)$ *valeurs.*

Démonstration. Pour choisir Δ, nous remarquons qu'il est inutile d'avoir une valeur Δ telle qu'aucun des coûts $C^*(I_j) + \Delta$ pour chaque I_j ne corresponde à un coût $C(A_i, I_j, m)$ pour tous les A_i. En effet, pour le problème de couverture d'ensemble, si F_i est l'ensemble correspondant à A_i, une instance I_j appartient à F_i si $C(A_i, I_j, m) \leq C^*(I_j) + \Delta$. Considérons une valeur Δ_1 tel qu'aucun coût $C^*(I_j) + \Delta$ pour chaque I_j ne corresponde à une valeur $C(A_i, I_j)$ pour tout A_i. Considérons aussi la plus grande valeur $\Delta_2 < \Delta_1$ telle qu'il y a une instance I_j et un



C	A_1	A_2	A_3	A_4	
I_1	1	5	2	5	
I_2	5	1	5	5	
I_3	5	2	5	2	
I_4	1	1	2	1	
I_5	2	5	1	5	

FIG. 4.3 – Illustration du choix de Δ. Avec la matrice de coûts, on note qu'en prenant toute valeur Δ dans $[1, 4[$, on a le même résultat qu'en prenant $\Delta = 1$.

algorithme A_i avec $C(A_i, I_j) = C^*(I_j) + \Delta_2$. Le problème de couverture d'ensembles avec Δ_2 est le même qu'avec Δ_1 étant donné que les ensembles F_i avec Δ_2 sont les même qu'avec Δ_1.

Cette observation nous amène à ne considérer que les valeurs Δ qui correspondent à une différence $C(A_i, I_j, m) - C^*(I_j)$ pour chaque I_j et A_i. Le nombre de telles valeurs est égal à nk. Aussi, pour trouver la bonne valeur de Δ, on peut procéder en calculant pour les nk valeurs de Δ le temps du **MA(Δ)** correspondant et en sélectionnant celui qui conduit au plus petit temps de résolution. ❑

La valeur optimale de Δ peut être choisie en temps polynomial. En outre, on peut facilement vérifier que l'algorithme **MA(Δ)** avec cette valeur aura un coût total de résolution au plus égal à celui de **MA**.
Jusqu'à présent nous nous sommes bornés à sélectionner des algorithmes dans le seul but de réduire la taille du sous ensemble d'algorithmes sur lequel nous partageons les ressources. Idéalement, il faudrait aussi que ce sous ensemble permette de résoudre plus vite les instances. Nous allons voir cela dans la suite.

Afin de tenir compte du coût de résolution des instances, nous proposons de modifier **MA(Δ)** en associant des poids w_i à chaque algorithme A_i. Nous prenons $w_i = \sum_{I_j \in F_i} C(A_i, I_j, m)$. Avec cette définition, chaque algorithme a comme poids le temps de résolution des instances qu'il peut résoudre à une distance Δ du coût de résolution optimal.
Nous redéfinissons $\mathcal{A}^*$ comme étant le sous ensemble d'algorithmes de poids minimal $\mathcal{A}^{w\Delta} \subseteq \mathcal{A}^*$ tel que si un algorithme $A_i \in \mathcal{A}^{w\Delta}$ alors il existe une instance I_j tel que $C(A_i, I_j, m) = C^*(I_j) + \Delta$. Sur le sous ensemble d'algorithmes $\mathcal{A}^{w\Delta}$, nous partageons alors les ressources en procédant comme dans **MA**. Nous obtenons ainsi un nouvel algorithme que nous notons **MA**$^w(\Delta)$. Les poids dans **MA**$^w(\Delta)$ sont définis de sorte à capturer la contribution de l'algorithme s'il est sélectionné dans le sous ensemble d'algorithmes $\mathcal{A}^{w\Delta}$. Cette borne n'est certainement pas précise mais nous permet au moins dans la sélection de marquer une préférence pour des algorithmes a priori moins coûteux.
On peut aisément établir que la sélection du sous ensemble d'algorithmes dans **MA**$^w(\Delta)$ correspond au problème de couverture d'ensembles avec des pondérations. Ce problème par ailleurs comme sa version non pondéré peut être approximé en temps polynomial. Enfin on peut noter

qu'avec **MA**$^w(\Delta)$**)**, on garde la même garantie d'approximation qu'avec **MA(**Δ**)**.

Les techniques de réduction d'algorithmes que nous avons proposées ci dessus peuvent aussi être étendues pour définir différents algorithmes d'allocation proportionnelle. Pour cela, il suffit de remplacer dans **MA**(Δ) et **MA**$^w(\Delta)$ le partage équitable de ressources par le partage proportionnel des ressources.

Nous avons proposé des heuristiques et un algorithme exact pour résoudre le *l-dRSSP*. Pour les heuristiques proposées, on a supposé que l'on a plus de ressources que d'algorithmes candidats. L'algorithme exact a une complexité exponentielle en le nombre d'algorithmes candidats disponibles. La section suivante est consacrée à l'évaluation expérimentale de ces algorithmes.

4.6 Expérimentations sur SAT

Nous appliquons nos différents algorithmes dans cette partie à la résolution du problème de satisfiabilité (SAT). Étant donné une formule en logique propositionnelle donnée comme une conjonction de clauses, le problème SAT consiste à dire si oui ou non celle ci est satisfiable pour au moins une interprétation des variables.

Pour nos expérimentations sur ce problème, nous utilisons une base de données SAT (SatEx [2]) contenant 23 solveurs SAT (algorithmes candidats) et un benchmark de 1303 instances du problème SAT. Pour chaque couple (algorithme, instance), cette base de données donne par ailleurs le temps d'exécution de l'instance sur l'algorithme en considérant une exécution sur une machine.

4.6.1 Les instances

Les 1303 instances de la base SatEx sont issus de plusieurs domaines d'application du problème SAT. Quelques uns de ces domaines sont : La logistique, la vérification formelle de microprocesseurs, l'ordonnancement. Ces instances sont aussi issues de plusieurs benchmarks pour SAT que l'on peut retrouver dans la compétition annuelle SAT la plus populaire [3]

4.6.2 Algorithmes

Les solveurs SAT contenus dans la base SatEx sont issus de trois familles principales [Simon and Chatalic 2001]. Ce sont :

– La famille des algorithmes DLL : $asat$, $csat$, $eqsatz$, $nsat$, $sat-grasp$, $posit$, $relsat$, $sato$, $sato-3.2.1$, $satz$, $satz-213$, $satz-215$, $zchaff$

[2] `http://www.lri.fr/~{}simon/satex/satex.php3`
[3] `http://www.satcompetition.org`

- La famille des algorithmes DP : $calcres$, dr, $zres$,
- La famille des algorithmes DLL randomisés : $ntab$, $ntab - back$, $ntab - back2$, $relsat - 200$
- Autres familles : $heerhugo$, $modoc$, $modoc - 2.0$

Nous renvoyons le lecteur à [Chatalic and Simon 2000] pour une connaissance approfondie de la typologie des algorithmes résolvant le problème SAT.

4.6.3 Plan d'expérimentation

Nous avons effectué 3 séries d'expérimentations.

1. L'objectif de la première série est de comparer le meilleur portfolio d'algorithmes avec le meilleur algorithme. Pour cela, nous avons sélectionné différents sous ensembles de $2, 3, 4, 5, 6$ algorithmes candidats sur lesquels nous avons déterminé le meilleur portfolio. Nous avons ensuite évalué le ratio entre le coût produit par ce portfolio dans chaque cas et le meilleur algorithme candidat. Pour chaque nombre d'algorithmes $2, 3...$, 30 tirages aléatoires ont été effectué. Dans les tests nous avons considéré $10, 15, 20, 25$ et 30 ressources à partager.
2. L'objectif de la seconde série d'expérimentations est de comparer la qualité de solution et du temps d'exécution de nos différentes algorithmes approchés. Nous avons considéré ici 100 ressources à partager et nous effectué 30 tirages de $5, 10, 15, 23$ algorithmes candidats.
3. Dans la dernière série, nous avons comparé nos algorithmes approchés avec la solution exacte. Nous avons considéré ici 30 tirages de $2, 3, 4, 5, 6$ algorithmes candidats avec 30 ressources à partager.

La base de données SatEx donne pour chaque algorithme candidat un coût d'exécution mesuré sur une machine. Nous avons supposé dans nos expérimentations que ce coût était chaque fois celui de l'algorithme candidat sur l'ensemble des ressources. Aussi, si $C_s(A_i, I_j)$ est le coût de l'algorithme A_i sur l'instance I_j dans SatEx, nous avons posé $C(A_i, I_j, m) = C_s(A_i, I_j)$ lorsque nous effectuions des expérimentations sur m ressources. Par hypothèse de linéarité, les autres coûts ont été déduit. Tous les jeux de données et les algorithmes que nous avons utilisés sont disponibles à l'adresse `http://moais.imag.fr/membres/yanik.ngoko/`.

4.6.4 Résultats

4.6.4.1 Première série d'expérimentations

La figure 4.4 présente le rapport entre le coût d'exécution du portfolio optimal et du meilleur algorithme candidat. Ce rapport est toujours inférieur à 1. Ceci est fort logique dans la mesure où l'affectation de toutes les ressources à un algorithme défini un portfolio qui n'est pas nécessairement optimal. Le fait toutefois que ce rapport soit dans la plupart des cas plus petit que 1 révèle l'intérêt de construire un portfolio d'algorithmes sur ce jeu de données.

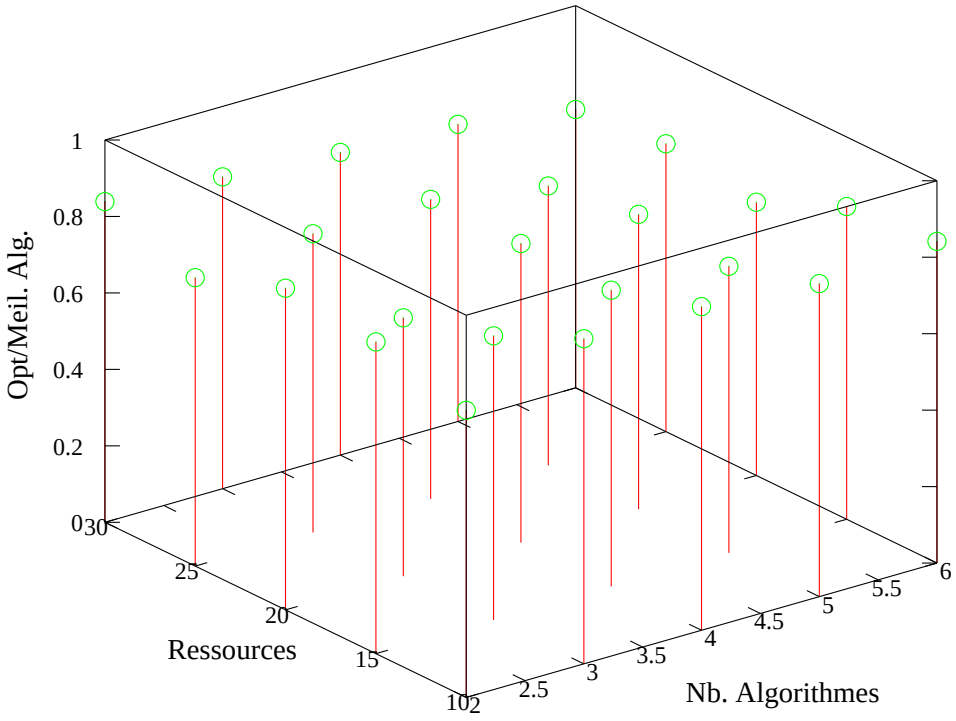


FIG. 4.4 – Comparaison du portfolio optimal avec avec l'algorithme candidat optimal.

4.6.4.2 Seconde série d'expérimentations

La figure 4.5 présente une comparaison des coûts des algorithmes issus de l'allocation moyenne et ceux issus de l'allocation proportionnelle. On a ici appliqué les techniques de sélection du plus petit sous ensemble couvrant d'algorithmes à l'allocation proportionnelle. On peut noter sur cette figure que le coût de l'allocation moyenne (**MA**) croît en fonction du nombre d'algorithmes. Ceci est dû au fait que l'augmentation du nombre d'algorithmes pour l'allocation moyenne implique nécessairement un plus grand partage des ressources. Si les algorithmes ajoutés ne sont pas extrêmement performants sur un nombre important d'instances alors la qualité de l'allocation moyenne est globalement détériorée. Ce résultat n'est pas satisfaisant car la qualité des résultats dans un portfolio d'algorithmes devrait pouvoir être améliorée par la connaissance d'un grand nombre d'algorithmes. Pour l'allocation proportionnelle (**PA**) la croissance des coûts d'exécution en fonction du nombre d'algorithmes n'est pas toujours observée car ce dernier prend en compte la possibilité d'avoir des algorithmes candidats dominants (ce qui a été le cas sur plusieurs jeu d'instances). En employant la sélection du plus petit sous ensemble couvrant d'algorithmes, on observe globalement sur l'allocation moyenne et proportionnelle une décroissance des coûts d'exécution. Ce résultat est satisfaisant et peut s'expliquer par le fait que la stratégie de sélection employée dans ce dernier cas fait en sorte que l'augmentation du nombre d'algorithmes accroît les possibilités d'avoir un ensemble couvrant d'instances qui soit réduit et moins coûteux en temps.

La figure 4.5 montre aussi que les algorithmes approchés issus de l'allocation moyenne sont

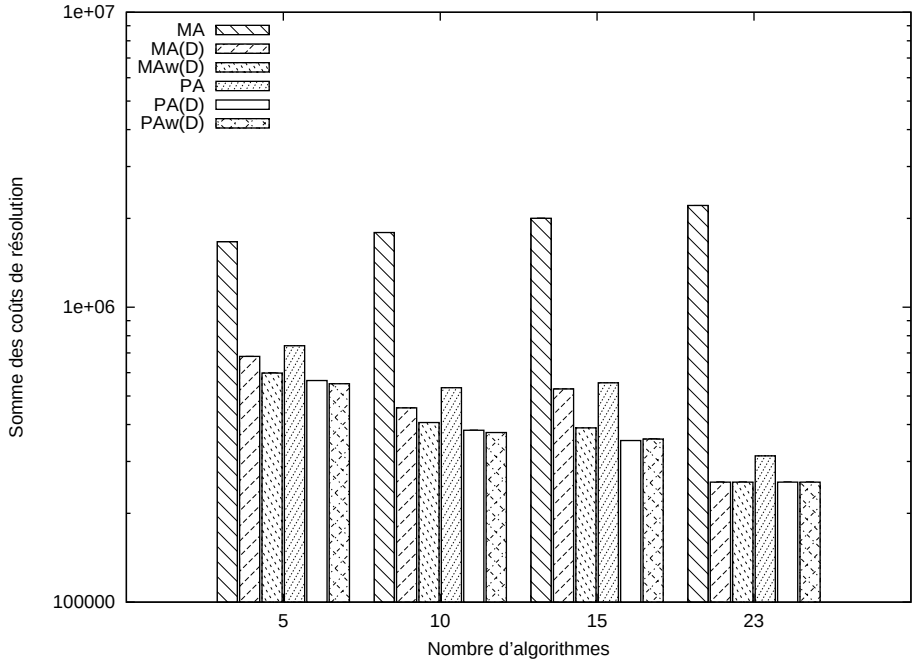


FIG. 4.5 – Coût d'exécution des algorithmes de l'allocation moyenne et proportionnelle sur 100 ressources.

en général moins bon que ceux issus de l'allocation proportionnelle. Nous pensons que ceci est lié au fait que l'allocation proportionnelle prend en compte la possibilité d'avoir des algorithmes qui dominent d'autres.

Dans la seconde série d'expérimentations, nous avons aussi mesure les temps d'exécutions de nos différents algorithmes approchés. Ces temps ont été mesuré sur un processeur Dual CPU E2160 @ 1.80GHz et sont présentés sur la figure 4.6.

On peut noter qu'en dehors des algorithmes approchés **MA** et **PA**, le temps d'exécution croît sensiblement pour tous les autres avec l'augmentation du nombre d'algorithmes candidats. Ceci confirme l'analyse théorique de la complexité des différents algorithmes.

4.6.4.3 Troisième série d'expérimentations

La figure 4.7 présente les ratio entre les coûts d'exécutions de nos algorithmes et la solution optimale.

Ce résultat montre que la qualité de **MA** est moins bonne quand on a plusieurs algorithmes. On observe aussi qu'en employant la technique de sélection des sous ensembles couvrant d'algorithmes, on reste près de la solution optimale. Par exemple sur 6 algorithmes candidats, lorsque l'allocation moyenne est à 4.04 de la solution optimale, **MA(Δ)** est à 1.52 et **PA(Δ)** à 1.27. On peut aussi noter que la stratégie de sélection avec pondération s'avère plus efficace

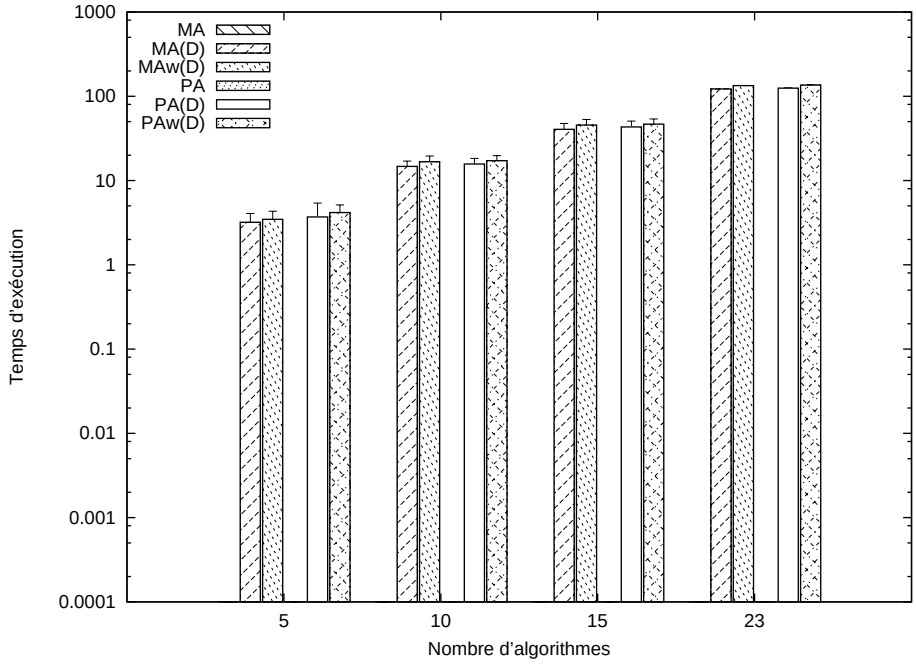


FIG. 4.6 – Temps d'exécution des algorithmes de l'allocation moyenne et proportionnelle sur 100 ressources.

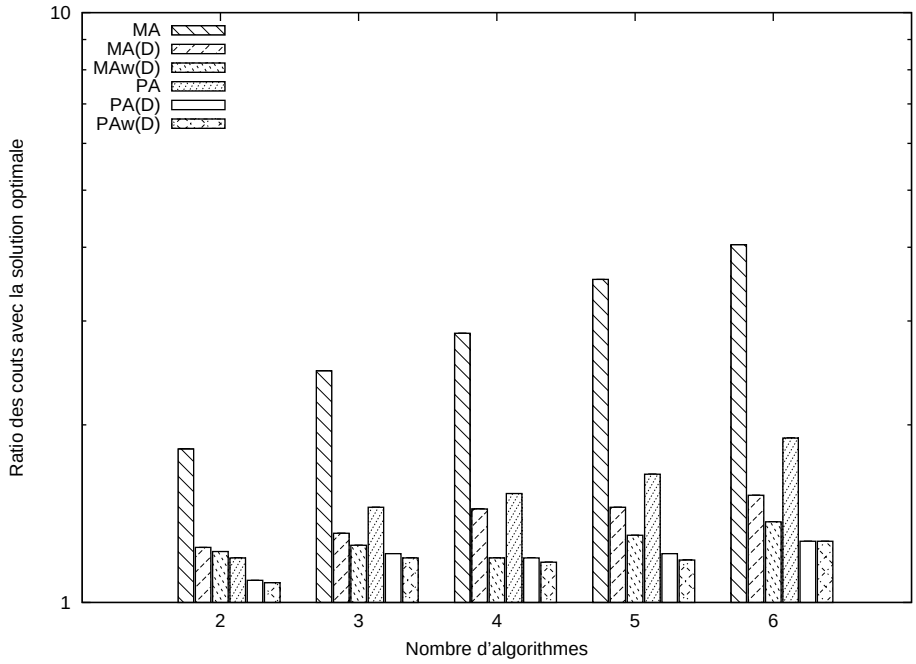


FIG. 4.7 – Ratio entre les coûts d'exécution et la solution optimale sur 30 ressources

que celle sans pondération. Enfin, la tendance précédemment observée sur la bonne qualité des algorithmes issus de l'allocation proportionnelle en comparaison de ceux issus de l'allocation moyenne est confirmée.

Dans le cadre de cette série, nous avons aussi comparé les temps d'exécutions entre algorithmes. Ces temps ont été mesure sur un processeur Dual CPU E2160 @ 1.80GHz. Les résultats sont présentés sur la figure 4.8

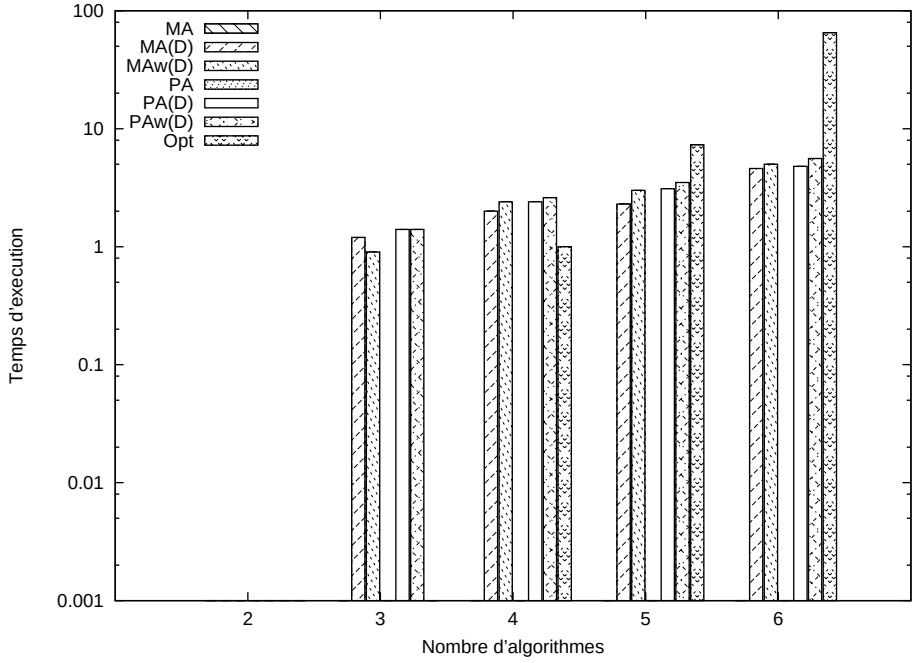


FIG. 4.8 – Temps d'exécution des algorithmes approchés et exacts sur 30 ressources

Ce résultat montre à nouveau que le temps d'exécution de la plupart des algorithmes croît en fonction de l'augmentation du nombre d'algorithmes candidats. On peut toutefois noter que l'algorithme exact est meilleur que les algorithmes approchés (**MA(**$\triangle$**)**, **PA(**$\triangle$**)**, **MA(**$^w\triangle$**)**, **PA(**$^w\triangle$**)**) lorsque le nombre d'algorithmes n'excède pas 4. A partir de 5 algorithmes candidats, le temps de l'algorithme exact dépasse celui des autres et a une croissance exponentielle. Ce phénomène est souvent courant sur les problèmes NP-Complet.

4.7 Conclusion

Dans ce chapitre nous avons montré comment on peut combiner statiquement des algorithmes à travers *l-dRSSP*. Dans cette approche, la même combinaison d'algorithmes est utilisée pour résoudre toutes les instances d'un problème car on extraie aucune information de l'instance en cours de résolution. Nous avons évalué la complexité du *l-dRSSP* et proposé des algorithmes exact et approchés pour le résoudre. A travers les expérimentations nous avons

évalué les solutions proposées. Nous avons validé ici l'idée selon laquelle l'utilisation de plusieurs algorithmes résolvant le même problème dans le cadre d'un portfolio peut réduire les coûts d'exécution et cela malgré la redondance des calculs inhérente au portfolio. Nous avons observé que la sélection d'un sous ensemble couvrant d'algorithmes permet dans la plupart des cas d'améliorer l'allocation moyenne et l'allocation proportionnelle. Par ailleurs même si sa garantie théorique est faible, l'allocation proportionnelle a l'avantage de s'adapter plus facilement à la présence d'algorithmes dominants dans le portfolio.
Nous envisageons les perspectives suivantes à l'issue de ce travail :

1. L'extension du *l-dRSSP* pour la prise en compte de la qualité des résultats produits par les algorithmes que l'on combine. Cette perspective est motivée par le fait que sur plusieurs problèmes d'optimisation, il existe des heuristiques conçues qui donnent différentes qualités de résultats en fonction du temps d'exécution. Dans ce cas, il est important de prendre en compte la qualité de la solution produite par la combinaison dans la minimisation du temps d'exécution. Quelques approches pour s'attaquer à ce problème ont été proposées dans [Petrik and Zilberstein 2006, Fukunaga 1999, Wu and Beek 2007]. Toutefois, les algorithmes qui y sont proposés se révèlent très heuristiques (garanties difficiles à obtenir) et exigeant en temps.
2. L'amélioration de la fonction de coût dans le *TSSP*. Nous ciblons ici la prise en compte du temps de sauvegarde et de changement de contexte dans le problème du partage de temps. Une approche pour cela consiste à fixer des points de reprise par algorithmes et de sauvegarde dans les algorithmes à partir desquels on peut facilement borner le surcoût lié à l'interruption de l'algorithme. Par exemple, il est plus facile de mesurer le coût des interruptions d'un algorithme qui n'est interrompu qu'en fin d'une boucle que celui d'un algorithme qui le serait en milieu d'une boucle. Cette fixation implique ensuite que les algorithmes ne seront pas nécessairement interruptibles à toute unité de temps.
3. L'extension de notre étude au contexte hétérogène. Ceci est en particulier motivée par développement massif des environnements hétérogène ces dernières années. Dans de tel environnements la variabilité des performances algorithmiques est souvent fréquente du fait des diverses configurations matérielles sur lesquels les algorithmes sont exécutés. Il est donc intéressant dans de tel contexte d'avoir des mécanismes efficaces de choix d'algorithmes.

Chapitre 5

Partage de ressources discret avec participation de tous les algorithmes

Résumé : *Dans ce chapitre nous étudions le problème de partage de ressources en supposant que tous les algorithmes doivent avoir au moins une ressource. Nous analysons la complexité de ce problème et nous montrons comment adapter les algorithmes proposés au chapitre précédent dans ce contexte. Nous proposons aussi une famille d'algorithmes pour approximer ce problème qui procède par combinaison des solutions partielles exactes et approchées.*

5.1 Introduction

Nous avons introduit au chapitre précédent le problème de partage de ressources (*dRSSP*). *dRSSP* permet de modéliser statiquement le choix de la meilleure combinaison algorithmique dans un contexte parallèle et homogène. Étant donné un problème cible $\mathcal{P}$, un ensemble fini d'instances $\mathcal{I}$ de ce dernier et un ensemble d'algorithmes candidats $\mathcal{A}$ le résolvant, l'on suppose ici que l'on a une connaissance du problème et des algorithmes à travers les coûts d'exécution de $\mathcal{I}$ sur $\mathcal{A}$. On cherche alors à adapter la répartition des ressources aux algorithmes en se plaçant dans un contexte où l'on suppose que les instances du problème ont un comportement similaire à celui d'une instance dans $\mathcal{I}$. Cette modélisation a un intérêt si l'ensemble $\mathcal{I}$ est représentative du problème cible.
dRSSP a été prouvé comme appartenant à la classe NP-complet. Nous avons vu au chapitre 4 qu'on peut le résoudre en utilisant un algorithme exact polynomial pour un nombre d'algorithmes fixé ou en employant des algorithmes approchés.

L'idée de la représentativité de l'ensemble $\mathcal{I}$ peut suggérer une condition portant sur la répartition des ressources dans toute solution au *dRSSP*. En effet, *dRSSP* est globalement motivé par le fait que pour un même problème cible, on peut trouver des algorithmes différents donnant différents temps d'exécution sur ses instances. Aussi, une condition pour la représentativité

de l'ensemble $\mathcal{I}$ impose qu'il comporte des instances montrant les avantages dans le temps d'exécution de chaque algorithme. A partir de là, l'on peut envisager que le meilleur partage de ressources construit à partir de $\mathcal{I}$ va inclure tous les algorithmes de $\mathcal{A}$ dans la répartition des ressources. En outre, nous avons proposé dans le chapitre 4 des techniques de sélection d'algorithmes qui permettaient de réduire le sous ensemble d'algorithmes que l'on a dans un problème de partage de ressources. Dès lors, la résolution du *dRSSP* peut être pensée en deux phases que sont : une phase de sélection des algorithmes suivie d'une phase de partage de ressources dans laquelle on affecte au moins une ressource à chaque algorithme. Enfin, nous présentons à la figure 5.1 une comparaison de la solution exacte dans laquelle on fait participer tous les algorithmes et le meilleur algorithme pris séparément. Dans cette figure pour chaque nombre d'algorithmes et de ressources, nous avons effectué 30 tirages d'algorithmes. Nous avons ensuite comparé le temps moyen de résolution de la solution optimale avec celui du meilleur algorithme.

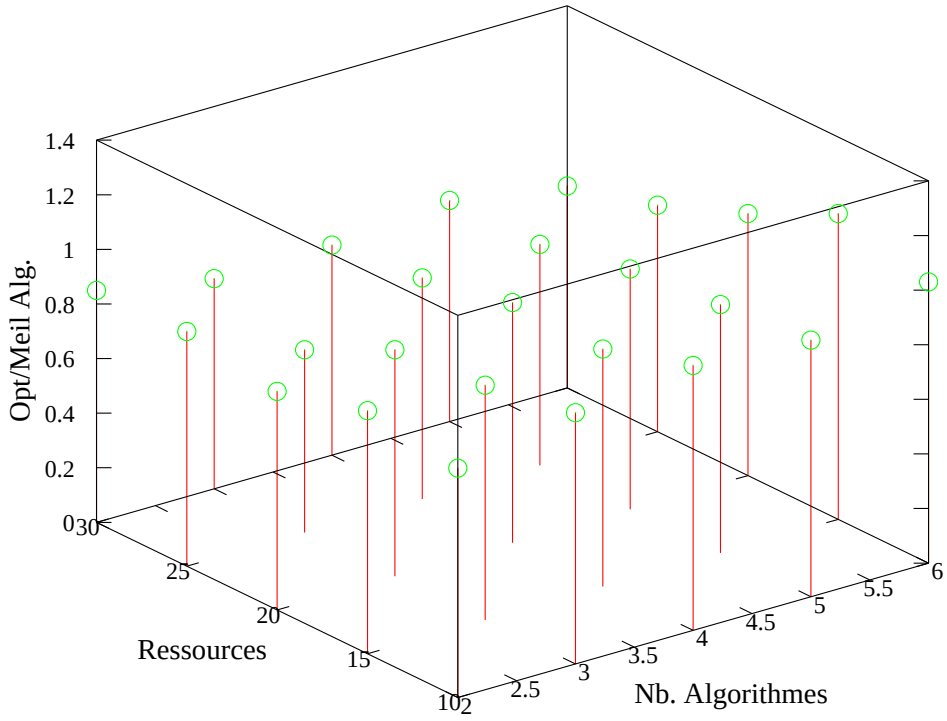


FIG. 5.1 – Comparaison de l'approche dans laquelle tous les algorithmes participent avec l'algorithme optimal.

On peut noter sur ce schéma que dans la plupart des cas, le portfolio optimal avec participation de tous les algorithmes offre un meilleur coût moyen d'exécution. Il existe certes des tirages sur lesquelles, ce constat n'est pas vrai [1]. En particulier ce phénomène apparaît quand le nombre d'algorithmes. Dans ces cas en effet, le fait de donner au moins une ressource à tout algorithme peut être pénalisant. Toutefois, si le nombre de ressources est important, ce phénomène n'est plus visible. Ensuite, on peut complètement éliminer le phénomène en prenant comme portfolio optimal, la meilleure solution obtenue en considérant le meilleur algorithme et le portfolio

[1] Les tirages de 6 algorithmes avec 15 ressources et les tirages de 6 algorithmes avec 10 ressources

optimal dans lequel chaque algorithme participe une fois. On peut noter qu'ainsi, la complexité du calcul du portfolio optimal garde le même ordre de grandeur.

Nous considérons dans ce chapitre le *r-dRSSP*, version réduite du *dRSSP* dans laquelle toute solution nécessite une affectation d'au moins une ressource à chaque algorithme candidat. On est ici dans une vision du problème de partage de ressources dans laquelle le partage optimal affecte toujours au moins une ressource à chaque algorithme. L'intérêt d'une telle hypothèse est difficile à prouver étant donné un problème de partage de ressources. Toutefois elle permet de *simplifier* a priori le problème de partage de ressources. En effet, nous avons établi la difficulté du *dRSSP* à partir des réductions au problème de couverture d'ensembles qui toutes impliquaient qu'une solution à ce dernier problème nécessite que l'on choisisse dans *dRSSP* le sous ensemble d'algorithmes auquel des ressources ne doivent pas être affectées.

Dans ce chapitre, nous nous focalisons sur l'étude du *r-dRSSP*. Nous analysons sa complexité et nous proposons des algorithmes pour le résoudre. Les résultats que nous présentons ici ont été publié dans [Bougeret et al. 2009a] puis une version augmentée dans [Bougeret et al. 2009b]. La suite du chapitre est organisée comme suit : Dans la Section 5.2, nous introduisons le *r-dRSSP* et nous analysons sa complexité ainsi que son approximabilité vis à vis du *dRSSP*. Dans la Section 5.3 nous revisitons les algorithmes basés sur l'allocation moyenne proposés dans le chapitre précédent. Dans la Section 5.4 nous introduisons une famille d'algorithmes basés sur la connaissance d'une solution partielle exacte du *r-dRSSP*. La section 5.5 présente les résultats expérimentaux obtenus par nos différents algorithmes et nous concluons à la Section 5.6.

5.2 Le problème de partage des ressources avec participation de tous les algorithmes

5.2.1 Définition

Nous nous plaçons dans un contexte où pour un problème à résoudre, nous disposons d'une liste finie d'algorithmes candidats $\mathcal{A}$ ainsi qu'un benchmark $\mathcal{I}$ d'instances représentatives du problème. Le problème de partage des ressources avec participation de tous les algorithmes peut être formulé ainsi :

restricted discrete Resource Sharing Scheduling Problem (r-dRSSP)

Instance : Un ensemble fini d'instances $\mathcal{I} = \{I_1, \ldots, I_n\}$, un ensemble fini d'algorithmes $\mathcal{A} = \{A_1, \ldots, A_k\}$, un ensemble de m ressources identiques, des coûts $C(A_i, I_j, p) \in R^+$ pour chaque $I_j \in \mathcal{I}$, $A_i \in \mathcal{A}$ et $p \in \{1, \ldots, m\}$, une valeur réelle $T \in R^+$.

Question : Existe t'il un vecteur $S = (S_1, \ldots, S_k)$ avec $S_i \in \{1, \ldots, m\}$ and $0 < \sum_{i=1}^{k} S_i \leq m$ tel que $\sum_{j=1}^{n} \min_{1 \leq i \leq k} \{C(A_i, I_j, S_i)\} \leq T$?

Comme nous l'avons déjà indiqué, le *r-dRSSP* est une adaptation du *dRSSP* dans laquelle on affecte au moins une ressource à chaque algorithme. Nous l'avons aussi défini en supposant que l'on est dans un contexte parallèle et homogène. Comme pour le *dRSSP*, nous admettrons ici l'hypothèse de linéarité indiquant que la performance des algorithmes soit proportionnelle au nombre de processeurs sur lesquels ils sont exécutés. Nous noterons *lr-dRSSP* pour désigner le *r-dRSSP* avec cette hypothèse. Dans la suite, nous analysons sa complexité.

5.2.2 Complexité

Théorème 5.2.1. lr-dRSSP *est NP-Complet.*

Démonstration. La preuve de ce résultat est inspirée de celle de la NP complétude du *dRSSP* dans le cas où l'on a plus de ressources que d'algorithmes.

Il est facile de vérifier que le *lr-dRSSP* est dans NP. Supposons qu'un vecteur $S = (S_1, \ldots, S_k)$ soit une solution candidate à une instance du *l-dRSSP* avec m ressources et k algorithmes. On peut vérifier en $O(nk)$ que S vérifie toutes les conditions d'une solution au problème en procédant comme suit : pour chaque instance I_j on trouve la valeur $v_j = \min\limits_{1 \leq i \leq k} \{\frac{C(A_i, I_j)}{S_i}\}$. Ceci peut être fait en $O(k)$. Pour les n instances on peut déterminer la somme des $v_j, j = 1, \ldots, n$ en $O(nk)$.

Nous allons procéder par une réduction au problème de couverture d'ensembles pour montrer que le *l-dRSSP* est NP complet.

On considère le problème de couverture d'ensembles suivant : Étant donné une valeur m et les ensembles finis $U = \{u_1, \ldots, u_n\}$, $F = \{F_1, \ldots, F_k\}$ avec $\bigcup_{i=1}^{k} F_i = U$, trouver un sous ensemble de $F^c \subseteq F$ tel que $\bigcup_{F_i \in F^c} = U$ et $|F^c| = x$. Pour le résoudre, nous allons utiliser *lr-dRSSP* comme suit :
On construit une instance du *lr-dRSSP* avec $m = x + k$ ressources et $n + k$ instances $\mathcal{I} = \{I_1, \ldots, I_{n+k}\}$, tel que chaque I_j, $j \leq n$ correspond à $u_j \in U$, $\mathcal{A} = \{A_1, \ldots, A_k\}$ avec A_i qui correspond à F_i. On fixe

$$C(A_i, I_j, m) = \begin{cases} \alpha > 0 & \text{si } u_j \in F_i \\ \beta = 2nm\alpha + 1 & \text{si } j = n + i \\ T + 1 & \text{sinon.} \end{cases}$$

et $T = nm\frac{\alpha}{2} + \beta(k - \frac{x}{2})$. On résout ce problème et si l'on obtient une solution, on prend l'ensemble $F^c = \{F_i \text{ t.q. } S_i \geq 2\}$ comme solution au problème de couverture d'ensembles. Le cas échéant, on répond que la couverture est impossible. Dans cette réduction, nous avons utilisé les instances $I_{n+1}, \ldots, I_{n+k}$ et fixé la valeur de β de sorte à avoir exactement x algorithmes ayant deux ressources dans toute solution au *lr-dRSSP* [2]. On peut facilement vérifier que l'ensemble F^c est bien une solution au problème de couverture d'ensembles.

[2] Voir annexe pour les détails sur T et β

Réciproquement, s'il existe une couverture d'ensemble F^c de taille m, alors le vecteur $S = (S_1, \ldots, S_k)$ où $\forall i, 1 \leq i \leq k\ S_i = \begin{cases} 2 & \text{si } F_i \in F^c \\ 1 & \text{sinon.} \end{cases}$ est une solution au problème du *lr-dRSSP* décrit plus haut. ❑

Corollaire 5.2.1. r-dRSSP *est NP complet*

Ce résultat indique que pour la version restreinte du *dRSSP*, il reste difficile de trouver un algorithme en temps polynomial à moins que $P = NP$.

Pour résoudre *lr-dRSSP*, on peut utiliser l'algorithme **ParcoursdRSSP** que l'on modifie en imposant au moins une ressource à chaque algorithme. On peut facilement déduire que la complexité de ce nouvel algorithme est en $O(p_k(m-k))$ où $p_k(m-k)$ est le nombre de partitions de l'entier m en exactement k parts. Cette complexité est exponentielle.
Une autre approche de résolution consiste à adapter les algorithmes que nous avons proposés pour *lr-dRSSP*. La section suivante est consacrée à une étude dans cette direction.

5.3 Allocation moyenne sur le *lr-dRSSP*

Dans cette section nous nous intéressons à l'adaptation sur *lr-dRSSP* des algorithmes de l'allocation moyenne proposés antérieurement. Nous commençons par introduire la borne inférieure que nous allons utiliser pour l'analyse des algorithmes.

5.3.1 Bornes inférieures

Pour trouver une borne inférieure au *lr-dRSSP*, on peut se baser sur le résultat suivant :

Propriéte 5.3.1. *Le temps moyen de résolution des instances $\mathcal{I}$ avec le* lr-dRSSP *est supérieur ou égal à celui pris pour le* l-dRSSP

Ce résultat est déduit du fait que toute solution au *lr-dRSSP* est une solution au *l-dRSSP*. Il suggère aussi que l'on pourrait construire une solution au *lr-dRSSP* par transformation d'une solution pour le *lr-dRSSP* afin de respecter la contrainte $S_i \geq 1\ \forall i \in \{1, \ldots, k\}$. Cette approche est en contradiction avec la motivation qui nous pousse à étudier *lr-dRSSP* car elle nous amène à résoudre à nouveau le *l-dRSSP*. De plus, la NP complétude du *l-dRSSP* rend difficile le calcul d'une telle borne.

Nous proposons ici de prendre comme borne inférieure l'expression
$Lb = \frac{m}{p_{max}} \sum_{j=1}^{n} \min_i \{C(A_i, I_j, m)\}$ définie au chapitre 4. On peut facilement éliminer p_{max} dans cette expression car $p_{max} \leq m - k - 1$.
La borne inférieure Lb est facilement atteignable en employant l'exemple 4.5.1 du chapitre 4

sur le *lr-dRSSP*.
Dans la suite nous revisitons les algorithmes basés sur l'allocation moyenne sur le *lr-dRSSP*.

5.3.2 Algorithme de l'allocation moyenne

Nous avons introduit pour résoudre le *l-dRSSP* des algorithmes basés sur une allocation moyenne des ressources sur le sous ensemble d'algorithmes $\mathcal{A}^* = \{A_i \in \mathcal{A} \text{ ssi } \exists j \, | C^*(I_j) = \min_i \{C(A_i, I_j, m)\}$. Dans ces algorithmes, il était important de parvenir à une sélection efficace des algorithmes participant au partage de ressources. Sur *lr-dRSSP*, la sélection des algorithmes tel que nous l'effectuons auparavant n'est plus justifiée car tous les algorithmes doivent participer au partage des ressources. Aussi, nous proposons avec l'algorithme $\mathbf{MA}^r$ une redéfinition de l'allocation moyenne. On peut facilement obtenir le résultat suivant :

Algorithm 3 MA^r(S)

```
1: q = m div k
2: r = m − k × q
3: for i = 1 to k do
4:         S_i = q
5: end for
6: for i = 1 to r do
7:         S_i = S_i + 1
8: end for
```

Proposition 5.3.1. ***MA***^{r} *est une* $(2k-1).\frac{p_{max}}{m}$ *approximation pour le* l-dRSSP *en* $O(k)$

Cette proposition suggère que la version du restreinte du *l-dRSSP* n'est pas mieux approximable car sur le *l-dRSSP*, nous avons obtenu un rapport d'approximation similaire. En disant cela, on ne prend néanmoins pas en compte le fait que la valeur de p_{max} dans le *lr-dRSSP* est dans le pire des cas plus petite que celle du *l-dRSSP*. Le résultat suivant permet de mieux cerner cela

Proposition 5.3.2. *Dans le pire des cas,* ***MA***^{r} *est une* k*-approximation pour* lr-dRSSP

Démonstration. Soit q le plus petit nombre de ressources affectées à un algorithme dans le *lr-dRSSP*. Étant donné une instance I_j dans la solution son temps d'exécution est $C(I_j) \leq \frac{m}{q} C^*(I_j)$. Aussi,

$$\begin{aligned} \frac{\sum_{j=1}^{n} C(I_j)}{Lb} &\leq \frac{(\frac{m}{q})\sum_{j=1}^{n} C^*(I_j)}{(\frac{m}{p_{max}})\sum_{j=1}^{n} C^*(I_j)} \leq \frac{p_{max}}{q} \\ &\leq \frac{m-k-1}{q} \leq \frac{qk+r-k-1}{q} \\ &\leq k + \frac{r-k-1}{q} \leq k \end{aligned}$$

❑

Dans le pire des cas, pour *l-dRSSP*, l'algorithme **MA** donne une solution qui est une $2k-1$ approximation. Aussi, on peut conclure que la contrainte d'affectation d'une ressource au moins à chaque algorithme nous permet d'avoir de meilleures approximation.

En utilisant la technique de sélection d'un sous ensemble d'algorithmes candidats, nous pensons qu'il est difficile de faire mieux que la borne de k dans le pire des cas. Dans la suite nous allons introduire une nouvelle technique qui procède par hybridation de $\mathbf{MA}^r$ et de l'algorithme exact pour obtenir de meilleures approximations sur *l-dRSSP*.

5.4 Approche hybride de résolution du *lr-dRSSP*

5.4.1 Idée générale

La difficulté du *lr-dRSSP* réside dans le choix de la bonne allocation des ressources. L'algorithme $\mathbf{MA}^r$ propose une allocation moyenne des ressources et donne une solution k approchée. Si cet algorithme a une bonne complexité en temps, il ne considère que très peu de partages parmi l'ensemble de ceux qui sont possibles. Pour améliorer l'approximation fournie par cet algorithme, une approche consiste à choisir un sous ensemble d'algorithmes candidats pour lesquels on considère l'ensemble des allocations de ressources possibles tout en appliquant le partage de ressources donné par l'algorithme $\mathbf{MA}^r$ sur le reste des algorithmes candidats. Dans cette approche, on fixe une valeur g et on scinde les algorithmes candidats en deux sous ensembles. Un sous ensemble $\mathcal{A}^+ = \{A'_1, \ldots, A'_g\}$ avec $A'_i, \in \mathcal{A}, \forall i \in \{1, \ldots, g\}$ des algorithmes sur lesquels on teste toutes les affectations de ressources possibles $(S_1, \ldots, S_g)$ avec $\sum_{i=1}^{g} S_i \leq m$ et un sous ensemble $\mathcal{A}^- = \{A'_{g+1}, \ldots, A'_k\}$, $A'_i, \in \mathcal{A}, \forall i \in \{g+1, \ldots, k\}$ des algorithmes sur lesquels on applique $\mathbf{MA}^r$ sur les $m - \sum_{i=1}^{g} S_i$ ressources restantes. Dans cette approche, on considère plus d'allocations de ressources que dans $\mathbf{MA}^r$. Son intérêt est que dans l'ensemble des allocations que l'on parcourt, on est sûr de tomber sur l'affectation des ressources donnée par l'algorithme exact sur les algorithmes dans $\mathcal{A}^+$. Dans ce cas, on ne se *trompe* que sur les allocations effectuées aux algorithmes restants (ceux dans $\mathcal{A}^- = \mathcal{A} \setminus \mathcal{A}^+$). Cette dernière remarque suggère que le choix du sous ensemble $\mathcal{A}^+$ est critique dans la mesure où l'on peut trouver pour eux l'allocation donnée par la solution optimale. Nous allons dans cette partie considérer deux approches pour un tel choix.

5.4.2 Choix quelconque des ressources

Une première approche pour choisir le sous ensemble $\mathcal{A}^+$ consiste à les prendre de façon aléatoire. Dans ce cas, nous avons l'algorithme $\mathbf{MA}^{G_1}$ pour *Mean Allocation with Guess 1*. En supposant que dans l'ensemble $\mathcal{A}^+$, on a les algorithmes $\{A_1, \ldots, A_g\}$, nous donnons ci dessous sa formulation.

Cet algorithme hybride l'approche exacte de résolution du *lr-dRSSP* avec l'algorithme de

Algorithm 4 MA^{G_1}(S, $index$, sum, $melCout$, Sol)

if $index = g + 1$ **then**
  for $j = g + 1$ to k **do**
    $S_i = \lfloor (m - sum)/(k - g) \rfloor$ {On partage les ressources restantes sur les $k - g$ algorithmes $A_g + 1, \ldots, A_k$}
  end for
  $Cout = Cout\_Portfolio(S)$ {On calcule le coût du portfolio induit par l'allocation S et on sauvegarde cela dans $Cout$}.
  if $Cout < melCout$ **then**
    $melCout = Cout$
    for $j = 1$ to k **do**
      $Sol_i = S_i$
    end for
  end if
else
  if $sum < m - (k - index + 1)$ **then**
    for $j = 1$ to $m - sum - (k - index + 1)$ **do**
      $S_{index} = j$
      MA^{G_1}(S, $index + 1$, $sum + j$, $melCout$, Sol)
    end for
  else { Toutes les ressources sont allouées, on avance $index$ pour calculer le coût de l'allocation}
    for $j = index$ to g **do**
      $S_j = 1$
    end for
    MA^{G_1}(S, $g + 1$, sum, $melCout$, Sol) {Ainsi on teste toutes les allocations possibles pour $A_1, \ldots, A_g$}
  end if
end if

l'allocation moyenne. Il doit être exécuté initialement avec $index = 1$, $sum = sol = 0$ et $melCout = +\infty$. La complexité et l'approximabilité de $\mathbf{MA}^{G_1}$ sont donnés par le résultat suivant :

Proposition 5.4.1. *$\mathbf{MA}^{G_1}$ donne une solution $k-g+1$ approchée du* lr-dRSSP *en $O((m+1)^g)$*

Démonstration. On peut noter que la complexité de cet algorithme est dominée par le nombre de partage possibles de la valeur m en g morceaux.
Pour la qualité de la solution de $\mathbf{MA}^{G_1}$, nous remarquons qu'en parcourant l'ensemble des allocations possibles, on tombe forcément sur une allocation $S = (S_1^{opt}, \ldots, S_g^{opt}, \ldots, S_k)$ où $S_1^{opt}, \ldots, S_g^{opt}$, sont les allocations de la solution optimale pour les algorithmes $A_1, \ldots, A_g$. Nous allons dans l'analyse borner le coût d'exécution de $\mathbf{MA}^{G_1}$ par celui obtenu dans cette allocation.
Désignons par $C(I_j)$ le coût de résolution de l'instance I_j sur l'allocation S et $C^{opt}(I_j)$ celui de l'algorithme optimal. Nous allons ici scinder l'ensemble des instances $\mathcal{I}$ en deux. Un sous ensemble $\mathcal{I}^-$ des instances qui sont résolus (dans la solution optimale) par les algorithmes dans $\mathcal{A}^+$ et le sous ensemble $\mathcal{I}^- = \mathcal{I} \setminus \mathcal{I}^+$. On a :

$$\begin{aligned}\frac{\sum_{j=1}^{n} C(I_j)}{\sum_{j=1}^{n} C^{opt}(I_j)} &\leq \frac{\sum\limits_{I_j \in \mathcal{I}^+} C(I_j) + \sum\limits_{I_j \in \mathcal{I}^-} C(I_j)}{\sum_{j=1}^{n} C^{opt}(I_j)} \\ &\leq \frac{\sum\limits_{I_j \in \mathcal{I}^+} C(I_j)}{\sum\limits_{I_j \in \mathcal{I}^+} C^{opt}(I_j)} + \frac{\sum\limits_{I_j \in \mathcal{I}^-} C(I_j)}{\sum\limits_{I_j \in \mathcal{I}^-} C^{opt}(I_j)}\end{aligned}$$

Or, si $I_j \in \mathcal{I}^+$ alors on sait que $C(I_j) \leq C^{opt}(I_j)$ car son coût de résolution serait au moins égal au plus petit coût donné par un algorithme dans $\mathcal{A}^+$. Aussi, on a

$$\frac{\sum_{j=1}^{n} C(I_j)}{\sum_{j=1}^{n} C^{opt}(I_j)} \leq 1 + \frac{\sum\limits_{I_j \in \mathcal{I}^-} C(I_j)}{\sum\limits_{I_j \in \mathcal{I}^-} C^{opt}(I_j)}$$

Posons $k' = k - g$ et $m' = m - \sum_{i=1}^{g} S_i$. En appliquant l'allocation moyenne pour partager les ressources à $\mathcal{A}^-$, on leur affecte chacun au moins q' ressources où q est obtenu par division euclidienne de m par k ($m' = qk' + r'$). Pour chaque instance $I_j \in \mathcal{I}^-$, soit $C^m(I_j)$ son plus petit temps de résolution avec un algorithme dans $\mathcal{A}^-$. On a $\forall I_j \in \mathcal{I}^-, C(I_j) \leq \frac{m}{q'} C^m(I_j)$. Étant donné que dans la solution optimale, on affecte au plus $m' - k' - 1$ ressources à un algorithme dans $\forall I_j \in \mathcal{I}^- C^{opt}(I_j) \geq \frac{m}{m'-k'-1} C^m(I_j)$. De ces deux inégalités, on déduit que

$$\begin{aligned}\frac{\sum_{j=1}^{n} C(I_j)}{\sum_{j=1}^{n} C^{opt}(I_j)} &\leq 1 + \frac{m'-k'-1}{q'} \leq 1 + \frac{q'k'+r'-k'-1}{q'} \\ &\leq 1 + k'\end{aligned}$$

❑

L'intérêt de ce résultat est que nous pouvons choisir différentes valeurs de g et avoir en un temps polynomial en g une solution $k-g+1$ approché du *lr-dRSSP*.
Nous avons vu qu'il est possible de trouver une meilleure approximation au *lr-dRSSP* en testant toutes les allocations possibles pour un nombre quelconque fixé d'algorithmes. Nous avons dans une première approche considéré le cas où les différents algorithmes pour lesquels on cherche l'allocation exacte sont pris de façon quelconque. Intuitivement toutefois, on peut noter qu'en considérant différent sous ensembles de g algorithmes, on obtient pas dans la solution finale de $\mathbf{MA}^{G_1}$. Nous allons dans la suite voir comment l'on peut améliorer le choix du sous ensemble $\mathcal{A}^+$

5.4.3 Prise en compte de tous les choix

Afin d'améliorer le choix du sous ensemble $\mathcal{A}^+$, nous proposons de prendre en compte tous les sous ensembles de g algorithmes candidats que l'on peut avoir. Pour cela, on répète l'algorithme $\mathbf{MA}^{G_1}$ sur toutes les combinaisons de g algorithmes que l'on peut avoir et la combinaison conduisant au plus petit temps d'exécution est retenue. Nous nommons ce nouvel algorithme $\mathbf{MA}^{G_2}$. La complexité et l'approximabilité de $\mathbf{MA}^{G_2}$ sont donnés par le résultat suivant :

Proposition 5.4.2. *$\mathbf{MA}^{G_2}$ donne une solution $\frac{k-1}{g}$ approchée du* lr-dRSSP *en $O(\binom{k}{g}(m+1)^g)$*

Démonstration. La complexité de cet algorithme peut facilement être déduite du fait que l'on choisis tous les sous ensembles de g algorithmes sur lesquels on applique $\mathbf{MA}^{G_1}$.
Pour analyser la qualité de la solution produite, nous procédons comme précédemment. Étant donné un algorithme A_i soit $\sigma(i)$ l'ensemble des instances dans la solution optimale sur lesquelles il est le meilleur. Soit $C^{opt}(A_i)$ sa participation dans le partage de ressource optimal pour la résolution des instances. Cette participation est obtenue en prenant la somme de ses coûts d'exécution sur les instances $\sigma(i)$ ($C^{opt}(A_i) = m \sum_{I_j \in \sigma(i)} \frac{C(A_i,I_j,m)}{S_i^{opt}}$). A travers son exécution, $\mathbf{MA}^{G_2}$ considère nécessairement un sous ensemble d'algorithmes $\mathcal{A}^+ = \{A'_1, \ldots, A'_g\}$ et une allocation de ressources S donnant le nombre optimal des ressources aux algorithmes $\mathcal{A}^+$ avec $C^{opt}(A'_1) \geq C^{opt}(A'_g) \geq C^{opt}(A'_{g+i})$, $i \geq 1$. Sans nuire à la généralité, nous allons ici supposer que $A'_i = A_i$. Soit $C(A_i)$ le coût total de résolution des instances $\sigma(i)$ avec l'allocation S ($C(A_i) = m \sum_{I_j \in \sigma(i)} \frac{C(A_i,I_j,m)}{S_i}$). Avec le choix d'algorithmes $\mathcal{A}^+$ et l'allocation S, si $T_{MA^{G_2}}$ est le coût nécessaire à la résolution des instances avec $\mathbf{MA}^{G_2}$, on a

$$\begin{aligned}
\frac{T_{MA^{G_2}}}{\sum_{i=1}^{k} C^{opt}(A_i)} &\leq \frac{\sum_{i=1}^{k} C(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)} \\
&\leq \frac{\sum_{i=1}^{g} C(A_i) + \sum_{i=g+1}^{k} C(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)} \\
&\leq \frac{\sum_{i=1}^{g} C^{opt}(A_i) + \sum_{i=g+1}^{k} C(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)}
\end{aligned}$$

Or, on a $C(A_i) \leq \frac{m}{S_i} \sum_{I_j \in \sigma(i)} C^*(I_j)$ et $C^{opt}(A_i) = \frac{m}{S_i^{opt}} \sum_{I_j \in \sigma(i)} C^*(I_j)$ ($C^*(I_j)$ est le plus petit coût de résolution possible de l'instance I_j). Aussi,

$$\begin{aligned}
\frac{T_{MA^{G_2}}}{\sum_{i=1}^{k} C^{opt}(A_i)} &\leq \frac{\sum_{i=1}^{g} C^{opt}(A_i) + \sum_{i=g+1}^{k} \frac{S_i^{opt}}{S_i} C^{opt}(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)} \\
&\leq \frac{\sum_{i=1}^{k} C^{opt}(A_i) + \sum_{i=g+1}^{k} (\frac{S_i^{opt}}{S_i} - 1) C^{opt}(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)} \\
&\leq 1 + \frac{\sum_{i=g+1}^{k} (\frac{S_i^{opt}}{S_i} - 1) C^{opt}(A_i)}{\sum_{i=1}^{k} C^{opt}(A_i)}
\end{aligned}$$

Si considère un partage équitable des ressources aux algorithmes $A_{g+1}, \ldots, A_k$ alors on a $\sum_{i=g+1}^{k} (\frac{S_i^{opt}}{S_i} - 1) \leq \frac{m'}{q'} - k'$. Ceci se déduit du fait que la somme de allocations $\sum_{i=g+1}^{k} S^{opt}$ est égale à m' et que que l'on a partagé équitablement les m' ressources restantes en q' aux algorithmes $A_{g+1}, \ldots, A_k$ On obtient alors

$$\begin{aligned}
\frac{T_{MA^{G_2}}}{\sum_{i=1}^{k} C^{opt}(A_i)} &\leq 1 + (\frac{m'}{q'} - k') \frac{C^{opt}(A_g)}{\sum_{i=1}^{k} C^{opt}(A_i)} \\
&\leq 1 + (\frac{m'}{q'} - k') \frac{C^{opt}(A_g)}{g C^{opt}(A_g)} \\
&\leq 1 + \frac{r'}{q'} \frac{1}{g} \\
&\leq 1 + \frac{k'-1}{g} \text{ car } r' < k' \text{ et } q' \geq 1
\end{aligned}$$

❏

En prenant en compte tous les sous ensembles possibles d'algorithmes, on peut donc établir théoriquement que l'on obtient une meilleure approximation de la solution optimale.
Les approches hybrides que nous avons introduit dans cette partie peuvent aussi être appliquées directement sur *l-dRSSP*. Néanmoins, nous précisons que dans ces cas, nous n'obtenons pas les mêmes rapports d'approximation. En particulier l'analyse est rendue difficile du fait que certains algorithmes peuvent ne pas avoir de ressources tant dans la solution optimale que dans les solutions que nous aurons en utilisant $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$.
Dans la section suivante, nous allons évaluer sur *lr-dRSSP* les différents algorithmes que nous avons présenté dans ce chapitre.

5.5 Expérimentations sur SAT

Pour les expérimentations, nous utilisons à nouveau la base SatEx considérée dans le chapitre 4. Nous rappelons que cette base donne les temps d'exécution de 1303 instances sur 23 solveurs. Ci dessous, nous présentons notre plan d'expérimentation.

5.5.1 Plan d'expérimentation

Nous avons effectué 2 séries d'expérimentations.

1.
2. L'objectif de la première série d'expérimentations est de comparer la qualité de solution et du temps d'exécution des algorithmes proposés dans ce chapitre. Nous avons considéré ici 50 ressources à partager entre les 23 algorithmes candidats.
3. Dans seconde série, nous avons comparé nos algorithmes approchés avec la solution exacte. Nous avons considéré ici 30 tirages de $5, 6$ algorithmes candidats avec 30 ressources à partager.

Tous les jeux de données et les algorithmes que nous avons utilisés sont disponibles à l'adresse `http://moais.imag.fr/membres/yanik.ngoko/`.

5.5.2 Résultats

5.5.2.1 Première série d'expérimentations

Sur la figure 5.2, nous présentons les coûts d'exécutions de nos algorithmes lorsqu'on fait varier g de 1 à 3. On peut noter que plus la valeur de g croît, meilleur est le portfolio produit. En outre l'algorithme $\mathbf{MA}^{G_2}$ est toujours meilleur que $\mathbf{MA}^{G_1}$ ce qui valide les rapports d'approximation établis pour ces algorithmes.

Sur la figure 5.3, nous présentons les temps d'exécution de $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$. On peut noter que le temps de $\mathbf{MA}^{G_2}$ est toujours plus grand que celui de $\mathbf{MA}^{G_1}$. Ceci est dû au fait que l'on considère un plus grand sous ensemble d'algorithmes candidats pour lesquels on parcourt l'ensemble des allocations possibles dans $\mathbf{MA}^{G_2}$.

5.5.2.2 Seconde série d'expérimentations

Sur la figure 5.4, nous présentons sur 30 ressources les coûts moyens d'exécution obtenus avec les algorithmes **MA**, $\mathbf{MA}^{G_1}$, $\mathbf{MA}^{G_2}$ et la solution optimale. On peut noter ici globalement (hormis sur l'allocation moyenne) que la qualité de la solution obtenue est meilleure quand on passe de 5 à 6 algorithmes. Ceci indique que les solutions proposées permettent effectivement de bénéficier de la complémentarité des algorithmes candidats.
Dans plusieurs cas, l'algorithme $\mathbf{MA}^{G_2}$ est très près de la solution optimale avec seulement $g = 2$. Ceci indique que cette approche est avantageuse dans le cas où pour le portfolio que l'on cible, il y a un sous ensemble d'algorithmes qui dans la solution optimale résout la plupart des instances. Cela explique aussi une autre observation à laquelle on aurait pu s'attendre : théoriquement, le rapport d'approximation des algorithmes $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ croît de façon linéaire en fonction du nombre d'algorithmes. En pratique toutefois, nous n'avons pas particulièrement noté cette observation comme dans le cas de l'algorithme **MA**. Nous pensons que

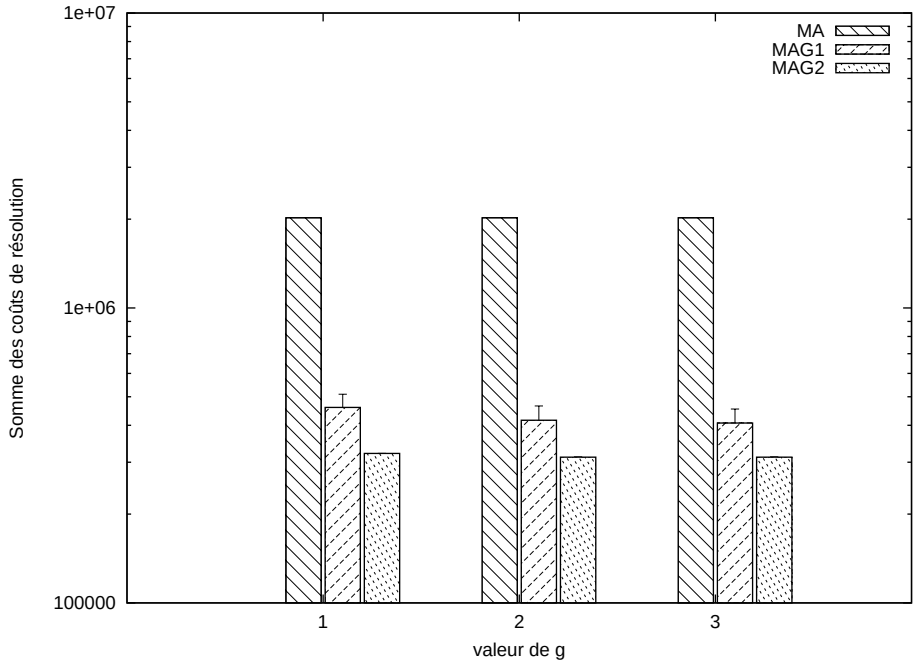


FIG. 5.2 – Coût d'exécution de $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ sur 100 ressources avec 23 algorithmes.

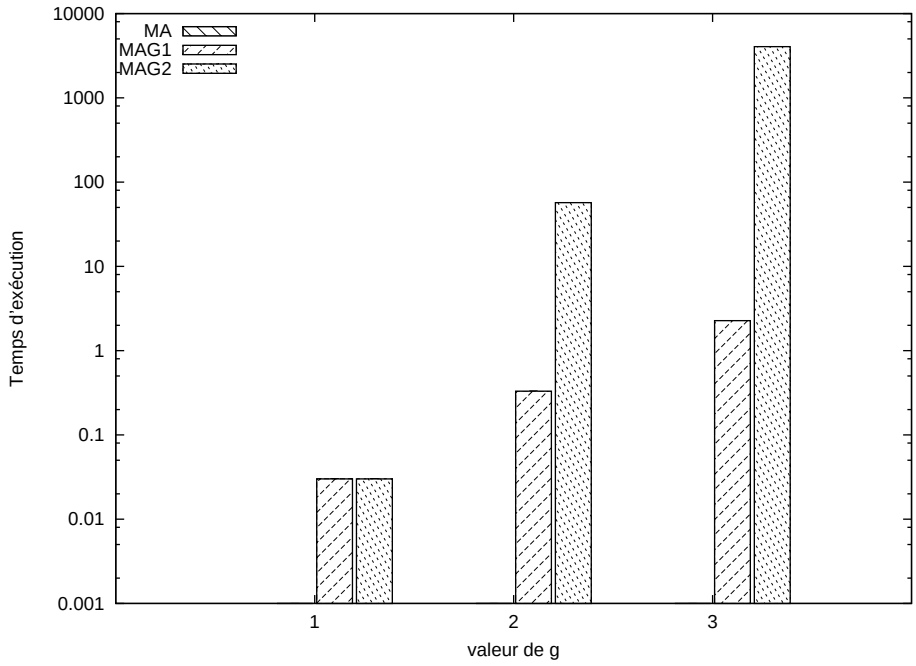


FIG. 5.3 – Temps d'exécution de $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ sur 100 ressources avec 23 algorithmes.

l'existence de sous ensemble d'algorithmes dominant dans la solution finale et repérés par les algorithmes $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ permet d'expliquer cela. En supposant en effet que l'essentiel des ressources dans un portfolio est concentré entre quelques algorithmes, si l'on trouve donc une allocation des ressources qui affecte la majeure partie des ressources à ces derniers en suivant éventuellement les disproportions existants dans l'affectation des ressources optimales à ceux ci, alors on peut être très près de l'optimal. On peut aussi noter sur cette figure que la performance de $\mathbf{MA}^{G_1}$ s'améliore quand la valeur de g croît. Cela confirme l'analyse théorique sur le rapport d'approximation de cet algorithme.

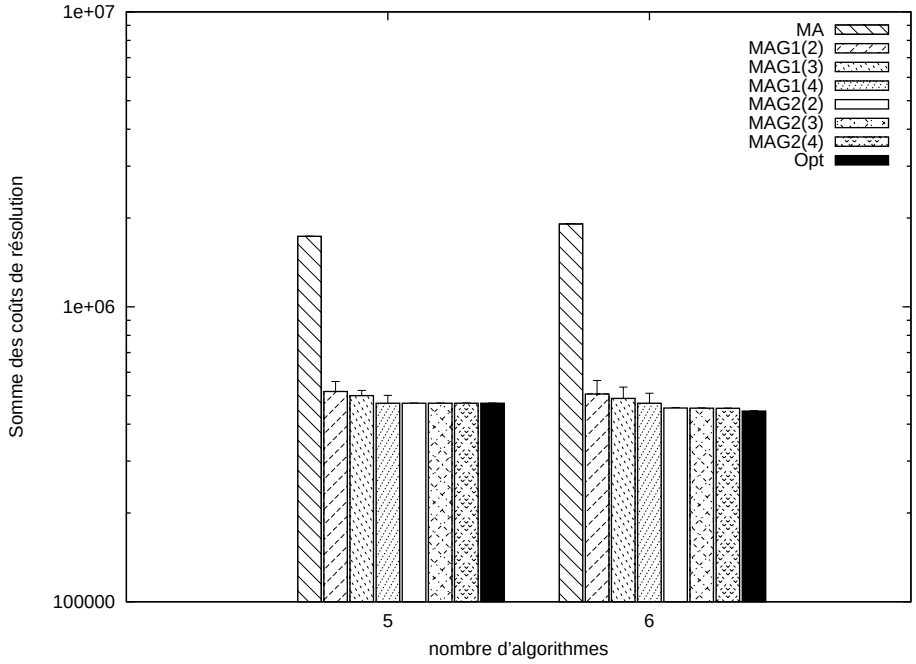


FIG. 5.4 – Coût d'exécution de $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ sur 30 ressources

Sur la figure 5.5, nous présentons les temps d'exécution obtenus dans cette série d'expérimentations. On note une fois de plus que le temps d'exécution de $\mathbf{MA}^{G_2}$ dépasse toujours celui de $\mathbf{MA}^{G_1}$. On peut aussi observer que lorsque la valeur de g est très grande, $\mathbf{MA}^{G_2}$ est beaucoup plus lent que l'algorithme optimal. Nous pensons que cela peut s'expliquer par l'implantation de $\mathbf{MA}^{G_2}$. Ici, il faut calculer les différents sous ensemble d'algorithmes candidats à considérer puis tester toutes les différentes allocations possibles sur ces derniers (dans l'algorithme candidat on a un seul sous ensemble candidat à considérer ce qui facilite l'implantation). En optimisant cette implantation, nous pensons que l'on peut obtenir un meilleur temps d'exécution.

5.6 Conclusion

Dans ce chapitre nous avons introduit une version restreinte du le *l-dRSSP*. Nous avons présenté des résultats de complexité sur ce problème et proposé des approches de résolution

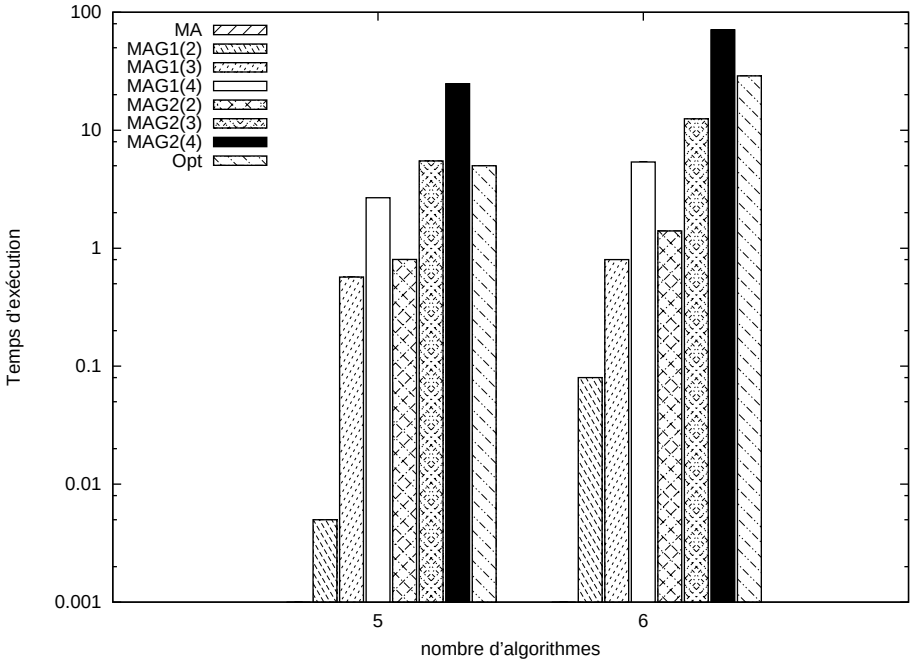


FIG. 5.5 – Temps d'exécution de $\mathbf{MA}^{G_1}$ et $\mathbf{MA}^{G_2}$ sur 30 ressources

(Celles ci peuvent par ailleurs être étendu au *l-dRSSP*). Les approches de résolution introduites dans ce chapitre combinent des solutions partielles exactes au problème avec des solutions approchées. Nous avons analysé la garantie théorique des différents algorithmes et nous les avons évalués expérimentalement en utilisant une base de données sur les solveurs SAT. Les expérimentations révèlent l'intérêt de considérer une sous solution exacte dans la construction des portfolio et la justesse des analyses théoriques effectuées. Par exemple $\mathbf{MA}^{G_1}$ n'a produit sur aucune expérimentation une solution de meilleure qualité que $\mathbf{MA}^{G_2}$. En outre, les expérimentations montrent que les approches hybrides proposées sont avantageuses lorsqu'on a un sous ensemble d'algorithmes dominants. Avec par exemple $\mathbf{MA}^{G_2}$, la connaissance du nombre exact de ressources pour seulement deux algorithmes dans plusieurs cas a permis l'obtenition du portfolio optimal.

Les bons résultats obtenus sur le *lr-dRSSP* suggèrent que dans le cas général, il serait intéressant de privilégier pour définir un partage de ressources une approche en deux phases. Dans un tel contexte, la première phase procède à une élimination de l'ensemble des algorithmes ne devant pas participer au partage. La seconde phase applique *l-dRSSP* sur l'ensemble des algorithmes restants.

A l'issue de ce travail, nous envisageons essentiellement d'étendre *lr-dRSSP* aux environnement hétérogènes où comme nous l'avons indiqué au chapitre précédent, la problématique du choix du meilleur algorithme est en général motivée. Nous envisageons aussi la parallélisation des approches hybrides et exactes proposées. Ceci est en particulier intéressant dans la mesure où nous définissons *lr-dRSSP* dans un contexte parallèle homogène. Nous envisageons enfin l'utilisation d'une technique de sélection d'un sous ensemble d'algorithmes peu coûteuse afin

d'optimiser $\mathbf{MA}^{G_1}$. L'approche employée dans $\mathbf{MA}^{G_2}$ est en effet coûteuse en temps lorsque le nombre d'algorithmes candidats dans $\mathcal{A}^+$ est important. On pourrait lui substituer celle employée au chapitre 4 basée sur le problème de couverture d'ensembles.

Chapitre 6

Conclusion

6.1 Bilan

Dans cette thèse, nous nous sommes intéressés à la combinaison des algorithmes résolvant un même problème avec pour objectif de réduire son temps de résolution. Nous nous sommes focalisés sur les solutions algorithmiques. Nous avons présenté les différentes modélisations que l'on peut adopter dans ce contexte avec en particulier : les techniques de sélection d'algorithmes, de combinaison récursive, d'apprentissage automatique et de portfolio d'algorithmes. Ces différentes techniques peuvent être regroupées en approches clairvoyantes et non clairvoyantes.
Nous avons choisi d'étudier plus spécifiquement les approches basées sur les portfolio d'algorithmes. Cette approche est intéressante lorsque l'on veut combiner les algorithmes sans modèle de performance analytique capable de donner une bonne estimation du temps d'exécution des algorithmes quelle que soit l'instance du problème à résoudre. L'approche des portfolio d'algorithmes exploite les éléments venant des approches on-line et celles basées sur l'apprentissage automatique. Au niveau des approches on-line, on exploite le modèle par portfolio que l'on adapte aux algorithmes. Au niveau des approches basées sur l'apprentissage, on exploite la notion de benchmark permettant d'inférer les mécanismes de choix.

Deux voies de construction des portfolio d'algorithmes ont été proposées dans cette thèse. La première est basée sur la méthode des plus proches voisins en apprentissage automatique. Cette approche est adaptative car elle procède en déterminant le portfolio le plus adapté pour chaque instance à résoudre. Celle ci a été appliquée au problème de résolution des systèmes linéaires où nous étions capable dans certaines situations de prédire 94% des cas un sous ensemble restreint contenant l'algorithme le mieux adapté pour résoudre le problème cible. Par ailleurs, nous avons pu obtenir de bons temps de résolution en moyenne des instances de systèmes linéaires.
Dans la seconde voie, on détermine le meilleur portfolio pour un problème uniquement à partir de la connaissance de ses performances sur un jeu de données que l'on suppose représentatif

du problème cible. L'objectif dans cette approche est de proposer une solution robuste qui doit en moyenne résoudre les instances du problème avec un temps d'exécution plus court que celui de n'importe quel algorithme pris individuellement. Dans la seconde approche, nous avons proposé un problème théorique pour la construction des portfolio d'algorithmes et analyser sa difficulté. Nous avons aussi proposé plusieurs solutions à ce problème que nous avons validées expérimentalement en simulant des portfolio pour la résolution du problème SAT.
Les expérimentations ont montré dans plusieurs situations que nos algorithmes pouvaient obtenir une solution qualitativement très près de l'optimale. En outre, elles ont montrées que nous étions capables d'obtenir de meilleur portfolio d'algorithmes en moyenne (un meilleur coût de résolution des instances) en utilisant un nombre plus important d'algorithmes candidats. Ce résultat est important car suggérant qu'avec une plus grande connaissance d'un problème (en termes d'algorithmes candidats le résolvant) on peut le résoudre plus efficacement.

Les résultats théoriques et expérimentaux que nous avons obtenus révèlent l'intérêt des approches par portfolio d'algorithmes dans un contexte non clairvoyant de combinaison des algorithmes. Nous précisons en particulier qu'il est important que l'on ait une grande variabilité dans les temps d'exécution pour que le surcoût lié à la redondance des calculs dans le portfolio soit amorti. Ceci est en général le cas avec les méthodes heuristiques de résolution des problèmes NP complet ou les algorithmes numériques itératifs. Par ailleurs, la représentativité du benchmark utilisé a un impact sur la qualité du portfolio d'algorithmes. Il est important de noter à cet effet que les éléments de la théorie de l'échantillonnage [Kearns and Vazirani 1994] permettent dans certains cas de caractériser la qualité des benchmarks. Ensuite, comme nous l'avons déjà souligné, pour certains problèmes bien connus notamment en optimisation combinatoire, il existe de nombreux travaux visant à élaborer des benchmarks de qualité. Les utiliser pour construire un portfolio d'algorithmes serait judicieux.

Dans cette thèse, nous avons employé deux approches de validation de nos solutions. Dans la première nous nous sommes situés dans un scénario où à partir d'un sous ensemble fini d'instances, on doit résoudre un ensemble d'instances beaucoup plus grand. Cette approche s'inspire de la technique du sous échantillonage en apprentissage automatique [Mitchell 1997] et reflète le fonctionnement en pratique de plusieurs bibliothèques bâties sur les
benchmarks [Whaley et al. 2001, Bilmes et al. 1997, Frigo and Johnson 1998]. La seconde approche de validation a consisté à construire un portfolio à partir d'un sous ensemble d'instances et de tester ce portfolio sur ce sous ensemble. A la différence de la première approche, celle ci est moins générique et est plus adaptée au cadre d'une compétition où l'on connaît à l'avance le comportement des instances que l'on aura sur les différents algorithmes. Cependant, elle n'est pas dénuée de tout fondement si l'on se situe dans un contexte où les instances à résoudre sont *très proches* d'un sous ensemble connu. De plus, le fait que l'on ne reconnaisse pas à chaque fois les instances que l'on résout dans la seconde approche nous permet de nous situer un peu dans le cas de la validation par sous échantillonage (on a un benchmark et un ensemble inconnus d'instances à résoudre).

6.2 Perspectives

Cette étude a permis de dégager une approche possible pour un problème difficile et souvent mal posé. Nous envisageons plusieurs perspectives. Elles portent principalement sur :

- L'amélioration de la validation
- Prise en compte des autres contextes d'exécution
- L'étude des stratégies pour le partage mixte du temps et des ressources
- La prise en compte de la qualité des résultats obtenus

6.2.1 Amélioration de la validation

Dans la validation sur les systèmes d'équations linéaires, nous avons ciblés la minimisation du nombre d'itérations et non du temps d'exécution. Si la proximité des coûts d'itérations entre les solveurs sélectionnés motive ce choix, il n'est toutefois pas entièrement fidèle au modèle de combinaison que nous avons proposé. En prenant en compte les temps d'exécution en particulier, on peut noter que les *effets de caches* apparaîtront dans la préemption des solveurs ce qui aura un impact sur les résultats. La même critique sur la validation peut être portée sur l'approche statique de combinaison que nous avons employé. L'usage des solveurs réels dans un contexte parallèle pourra faire apparaître d'autres considérations importantes que nous n'avons pas relevé.

6.2.2 Prise en compte des autres contextes d'exécution

Dans ce manuscrit, nous avons considéré uniquement les environnements parallèles homogènes en considérant un parallélisme idéal. Cette étude doit être étendue aux cas des environnements hétérogènes et distribués. Dans ces cas, les modèles de performance que nous avons proposés sur le parallélisme des algorithmes ne seraient pas nécessairement valides. Ceci nécessitera sans doute de revisiter les solutions proposées.

6.2.3 Partage du temps et des ressources

Dans cette étude, nous avons considéré séparément la construction des portfolio par partage de temps et par partage de ressources. On peut aussi envisager de construire des modèles par portfolio dans lesquels on met en oeuvre simultanément les deux types de partage. Le principal intérêt de ce choix est d'accroître la combinatoire possible dans la combinaison des algorithmes. Dans cette perspective toutefois, il serait intéressant de proposer au préalable un modèle efficace pour la mesure du temps dans le partage des ressources qui prennent en compte les effets de la préemption des algorithmes (qui nécessite de sauvegarder le contexte et de redéployer les données).

6.2.4 Prise en compte de la qualité des résultats obtenus

Dans cette étude, nous nous sommes focalisés à la combinaison des algorithmes afin de réduire uniquement le temps d'exécution. Certains algorithmes toutefois notamment dans la résolution des systèmes linéaires ou sur le problème SAT proposent différentes qualité de solution en fonction du temps d'exécution. Par exemple dans la première série d'expérimentations réalisées au chapitre 3, nous avons noté dans certains cas que seuls 57% des systèmes linéaires étaient effectivement résolus. La prise en compte de la qualité des résultats devient ainsi un facteur important. Une approche dans cette optique a été proposée dans [Fukunaga 1999]. Elle propose de fixer un temps d'exécution maximal pour l'obtention d'une solution et de déterminer la combinaison conduisant à la meilleure qualité de solution possible. De façon générale, les techniques d'optimisation dans un contexte multi-objectif [Saule 2008] pourraient ici être envisagées en déterminant les meilleurs compromis entre la qualité d'une solution et le temps qu'il a fallu pour l'obtenir.

Bibliographie

[Agarwal et al. 1994] Agarwal, R. C., Gustavson, F. G., and Zubair, M. (1994). A high-performance matrix-multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Res. Dev.*, 38(6) :673–681.

[An et al. 2003] An, P., Jula, A., Rus, S., Saunders, S., Smith, T., Tanase, G., Thomas, N., Amato, N., and Rauchwerger, L. (2003). STAPL : An adaptive, generic parallel C++ library. *Lecture notes in computer science*, pages 193–208.

[Auer et al. 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3) :235–256.

[Auer et al. 2003] Auer, P., Cesa-Bianchi, N., Freund, Y., and Schapire, R. E. (2003). The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1) :48–77.

[Barrett et al. 1994] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. (1994). *Templates for the Solution of Linear Systems : Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

[Barrett et al. 1996] Barrett, R., Berry, M., Dongarra, J., Eijkhout, V., and Romine, C. (1996). Algorithmic bombardment for the iterative solution of linear systems : a poly-iterative approach. *J. Comput. Appl. Math.*, 74(1-2) :91–109.

[Berman et al. 2001] Berman, F., Chien, A., Cooper, K., Dongarra, J., Foster, I., Gannon, D., Johnsson, L., Kennedy, K., Kesselman, C., Mellor-Crumme, J., Reed, D., Torczon, L., and Wolski, R. (2001). The grads project : Software support for high-level grid application development. *Int. J. High Perform. Comput. Appl.*, 15(4) :327–344.

[Bhowmick et al. 2002] Bhowmick, S., Raghavan, P., and Teranishi, K. (2002). A combinatorial scheme for developing efficient composite solvers. In *International Conference on Computational Science (2)*, pages 325–334.

[Bhowmick et al. 2006] Bhowmick, S., V.Eijkhout, Y.Freund, Fuentes, E., and Keyes, D. (2006). Application of machine learning to the selection of sparse linear solvers. `www.tacc.utexas.edu/~eijkhout/Articles/2006-bhowmick.pdf`.

[Bida and Toledo 2006] Bida, E. and Toledo, S. (2006). An automatically-tuned sorting library. Technical report, School of Computer Science, Tel-Aviv university.

[Bilmes et al. 1997] Bilmes, J., Asanovic, K., Chin, C.-W., and Demmel, J. (1997). Optimizing matrix multiply using PHiPAC : a portable, high-performance, ANSI C coding methodology.

In *ICS '97 : Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA. ACM.

[Blackford et al. 1997] Blackford, L. S., Choi, J., Cleary, A., D'Azeuedo, E., Demmel, J., Dhillon, I., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R. C., and (editor), J. J. D. (1997). *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[Boddy and Dean 1994] Boddy, M. and Dean, T. L. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artif. Intell.*, 67(2) :245–285.

[Borodin and El-Yaniv 1998] Borodin, A. and El-Yaniv, R. (1998). *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA.

[Bougeret et al. 2009a] Bougeret, M., Dutot, P., Goldman, A., Ngoko, Y., and Trystram, D. (2009a). Combining multiple heuristics on discrete resources. In *11th Workshop on Advances in Parallel and Distributed Computational Models APDCM, (IPDPS)*.

[Bougeret et al. 2009b] Bougeret, M., Dutot, P.-F., Goldman, A., Ngoko, Y., and Trystram, D. (2009b). Combining Multiple Heuristics on Discrete Resources. *International Journal on Foundation of Computer Science*, page to appear.

[Buisson et al. 2005] Buisson, J., André, F., and Pazat, J.-L. (2005). A framework for dynamic adaptation of parallel components. In *PARCO*, pages 65–72.

[Chatalic and Simon 2000] Chatalic, P. and Simon, L. (2000). Zres : The old davis-putman procedure meets zbdd. In *CADE*, pages 449–454.

[Chen et al. 2003] Chen, Z., Dongarra, J., Luszczek, P., and Roche, K. (2003). Self-adapting software for numerical linear algebra and lapack for clusters. *Parallel Comput.*, 29(11-12) :1723–1743.

[Coleman 2006] Coleman, B. (2006). Quality vs. performance in lookahead scheduling. In *JCIS*.

[Cormen et al. 2001] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms*. The MIT press.

[Culler et al. 1996] Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Santos, E. E., Schauser, K. E., Subramonian, R., and von Eicken, T. (1996). Logp : A practical model of parallel computation. *Commun. ACM*, 39(11) :78–85.

[Cung et al. 2006] Cung, V. D., Danjean, V., Dumas, J.-G., Gautier, T., Huard, G., Raffin, B., Rapine, C., Roch, J.-L., and Trystram, D. (2006). Adaptive and hybrid algorithms : Classification and illustration on triangular system solving. In *Transgressive Computing TC'2006*, pages 112–117, Granada, Spain. JG Dumas Editor.

[Demmel et al. 2006] Demmel, J., Dongarra, J., Parlett, B. N., Kahan, W., Gu, M., Bindel, D., Hida, Y., Li, X. S., Marques, O., Riedy, E. J., Vömel, C., Langou, J., Luszczek, P., Kurzak, J., Buttari, A., Langou, J., and Tomov, S. (2006). Prospectus for the next LAPACK and ScaLAPACK libraries. In *PARA*, pages 11–23.

[Dongarra et al. 2006a] Dongarra, J., Bosilca, G., Chen, Z., Eijkhout, V., Fagg, G., Fuentes, E., Langou, J., Luszczek, P., Pjesivac-Grbovic, J., Seymour, K., et al. (2006a). Self-adapting numerical software (SANS) effort. *IBM Journal of Research and Development*, 50(2-3) :223–238.

[Dongarra et al. 2006b] Dongarra, J., Bosilca, G., Chen, Z., Eijkhout, V., Fagg, G. E., Fuentes, E., Langou, J., Luszczek, P., Pjesivac-Grbovic, J., Seymour, K., You, H., and Vadhiyar, S. S. (2006b). Self-adapting numerical software (SANS) effort. *IBM J. Res. Dev.*, 50(2/3) :223–238.

[Dongarra et al. 2003] Dongarra, J., Luszczek, P., and Petitet, A. (2003). The linpack benchmark : past, present and future. *Concurrency and Computation : Practice and Experience*, 15(9) :803–820.

[Ern et al. 1994] Ern, A., Giovangigli, V., Keyes, D. E., and Smooke, M. D. (1994). Towards polyalgorithmic linear system solvers for nonlinear elliptic problems. *SIAM J. Sci. Comput.*, 15(3) :681–703.

[Feige 1998] Feige, U. (1998). A threshold of ln for approximating set cover. *J. ACM*, 45(4) :634–652.

[Fink 1998] Fink, E. (1998). How to solve it automatically : Selection among problem-solving methods. In *proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 128–136. AAAI Press.

[Frigo and Johnson 1998] Frigo, M. and Johnson, S. G. (May 1998). FFTW : An adaptive software architecture for the FFT. In *proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Seattle, Washington. ACM SIGARC.

[Fukunaga 1999] Fukunaga, A. S. (1999). Portfolios of genetic algorithms. In *GECCO*, page 786.

[Gagliolo and Schmidhuber 2006] Gagliolo, M. and Schmidhuber, J. (2006). Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3–4) :295–328. AI&MATH 2006 Special Issue.

[Gagliolo and Schmidhuber 2008] Gagliolo, M. and Schmidhuber, J. (2008). Algorithm selection as a bandit problem with unbounded losses. *CoRR*, abs/0807.1494.

[Gannon et al. 2000] Gannon, D., Bramley, R., Stuckey, T., Villacis, J. B. J., Akman, E., Berg, F., Diwan, S., and Govindaraju, M. (2000). *The Linear System Analyzer*. In E.N. Houstis, J.R. Rice, E. Gallopoulos, and R. Bramley, editors, Kluwer Academic Publishers :Dordrecht.

[Garey et al. 1979] Garey, M., Johnson, D., Backhouse, R., von Bochmann, G., Harel, D., van Rijsbergen, C., Hopcroft, J., Ullman, J., Marshall, A., Olkin, I., et al. (1979). *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Springer.

[Gebruers et al. 2005] Gebruers, C., Hnich, B., Bridge, D. G., and Freuder, E. C. (2005). Using cbr to select solution strategies in constraint programming. In *ICCBR*, pages 222–236.

[Gomes and Selman 2001] Gomes, C. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1) :43–62.

[Goto and van de Geijn 2008] Goto, K. and van de Geijn, R. A. (2008). Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3) :1–25.

[Grayson and van de Geijn 1996] Grayson, B. and van de Geijn, R. A. (1996). A high performance parallel strassen implementation. *Parallel Processing Letters*, 6(1) :3–12.

[Guo 2003] Guo, H. (2003). *Algorithm selection for sorting and probabilistic inference : a machine learning-based approach*. PhD thesis, Kansas State University, Manhattan, KS, USA. Major Professor-Hsu, William H.

[Hochbaum 1997] Hochbaum, D. S. (1997). *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, Department of Industrial Engineering.

[Houstis et al. 2000] Houstis, E. N., Catlin, A. C., Rice, J. R., Verykios, V. S., Ramakrishnan, N., and Houstis, C. E. (2000). PYTHIA-II : a knowledge/database system for managing performance data and recommending scientific software. *ACM Trans. Math. Softw.*, 26(2) :227–253.

[Huberman et al. 1997] Huberman, B., Lukose, R., and Hogg, T. (1997). An economics approach to hard computational problems. *Science*, 275(5296) :51.

[Hunold et al. 2004] Hunold, S., Rauber, T., and Rünger, G. (2004). Multilevel hierarchical matrix multiplication on clusters. In *ICS '04 : Proceedings of the 18th annual international conference on Supercomputing*, pages 136–145, New York, NY, USA. ACM.

[Kamath and Weeratunga 1990] Kamath, C. and Weeratunga, S. (1990). Implementation of two projection methods on a shared memory multiprocessor : Dec vax 6240. *Parallel Computing*, 16(2-3) :375–382.

[Kearns and Vazirani 1994] Kearns, M. J. and Vazirani, U. V. (1994). *An Introduction to Computational Learning Theory*. MIT Press.

[Kelley 1995] Kelley, C. (1995). *Iterative methods for Linear and Nonlinear Equations*. SIAM, Philadelphia, PA.

[Knuth 1975] Knuth, D. E. (1975). Estimating the efficiency of bactratck problems. *Mathematics of Computation*, 29(129) :121–136.

[Knuth 1998] Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3 : Sorting and Searching*. Addison-Wesley second Edition.

[Kuefler and Chen 2008] Kuefler, E. and Chen, T.-Y. (2008). On using reinforcement learning to solve sparse linear systems. In *ICCS (1)*, pages 955–964.

[Lagoudakis and Littman 2000] Lagoudakis, M. and Littman, M. (2000). Algorithm selection using reinforcement learning. In *proceedings of the Seventeenth International Conference on Machine Learning*, pages 511–518, San Francisco, CA. Morgan Kaufmann.

[Lamarca and Ladner 1999] Lamarca, A. and Ladner, R. (1999). The influences of caches on the performance of sorting. *Journal of Algorithms*, 31(1) :66–104.

[Leung et al. 2004] Leung, J., Kelly, L., and Anderson, J. H. (2004). *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA.

[Li et al. 1997] Li, J., Skjellum, A., and Falgout, R. D. (1997). A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. *Concurrency - Practice and Experience*, 9(5) :345–389.

[Li et al. 2004] Li, X., Garzarán, M. J., and Padua, D. (2004). A dynamically tuned sorting library. In *International Symposium on Code Generation and Optimization (CGO)*, pages 111–124.

[Lobjois and Lemaître 1998] Lobjois, L. and Lemaître, M. (1998). Branch and bound algorithm selection by performance prediction. In *AAAI '98/IAAI '98 : Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 353–358, Menlo Park, CA, USA. American Association for Artificial Intelligence.

[Markowitz 1999] Markowitz, H. (1999). The early history of portfolio theory : 1600-1960. *Financial Analysts Journal*, pages 5–16.

[McCracken et al. 2003] McCracken, M. O., Snavely, A., and Malony, A. D. (2003). Performance modeling for dynamic algorithm selection. In *International Conference on Computational Science*, pages 749–758.

[McGeoch et al. 2000] McGeoch, C. C., Sanders, P., Fleischer, R., Cohen, P. R., and Precup, D. (2000). Using finite experiments to study asymptotic performance. In *Experimental Algorithmics*, pages 93–126.

[Mitchell 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill International Edit.

[Nachtigal et al. 1992] Nachtigal, N. M., Reddy, S. C., and Trefethen, L. N. (1992). How fast are nonsymmetric matrix iterations. *SIAM J. Matrix Anal. Appl.*, 13(3) :778–795.

[Nasri and Trystram 2004] Nasri, W. and Trystram, D. (2004). A poly-algorithmic approach applied for fast matrix multiplication on clusters. In *IPDPS*.

[Ngoko 2006] Ngoko, Y. (2006). Poly-algorithmes pour une programmation efficace des problèmes numériques. exemple du produit de matrices. Mémoire de DEA.

[Ngoko and Trystram 2009a] Ngoko, Y. and Trystram, D. (2009a). Combining numerical iterative solvers. In *PARCO*, pages 43–50.

[Ngoko and Trystram 2009b] Ngoko, Y. and Trystram, D. (2009b). Combining SAT solvers on dicrete resources. In *HPCS, International Conference on high performance computing and simulation*, pages 153–160, Leipzig, Germany.

[O'Mahony et al. 2008] O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., and O'Sullivan, B. (2008). Using case-based reasoning in an algorithm portfolio for constraint solving. In *Proceedings of the 19th Irish Conference on Artificial Intelligence and Cognitive Science*.

[Petrik and Zilberstein 2006] Petrik, M. and Zilberstein, S. (2006). Learning static parallel portfolios of algorithms. In *Ninth International Symposium on Artificial Intelligence and Mathematics*.

[Rice 1979] Rice, J. R. (1979). Methodology for the algorithm selection problem. *Performance evaluation of numerical software L D Dosdisk Ed, North Holland, Amsterdam*, pages 301–307.

[Roch et al. 2006] Roch, J.-L., Traoré, D., and Bernard, J. (2006). On-line adaptive parallel prefix computation. In *Euro-Par*, pages 841–850.

[Saad 2003] Saad, Y. (2003). *Iterative methods for sparse linear systems*. Society for Industrial Mathematics.

[Saule 2008] Saule, E. (2008). *Algorithmes d'approximation pour l'ordonnancement multi-objectif. Application aux systèmes parallèles et embarqués*. PhD thesis, Institut National Polytechnique de Grenoble, Grenoble, France.

[Sayag et al. 2006] Sayag, T., Fine, S., and Mansour, Y. (2006). Combining multiple heuristics. In Durand and Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *LNCS*, pages 242–253. Springer.

[Simon and Chatalic 2001] Simon, L. and Chatalic, P. (2001). SATEx : a web-based framework for SAT experimentation. *Electronic Notes in Discrete Mathematics*, 9 :129–149.

[Streeter et al. 2007] Streeter, M., Golovin, D., and Smith, S. (2007). Combining multiple heuristics online. In *proceedings of the national conference on artificial intelligence*, volume 22, page 1197. Menlo Park, CA ; Cambridge, MA ; London ; AAAI Press ; MIT Press ; 1999.

[Whaley et al. 2001] Whaley, R. C., Petitet, A., and Dongarra, J. (2001). Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2) :3–35.

[Wu and Beek 2007] Wu, H. and Beek, P. v. (2007). On portfolios for backtracking search in the presence of deadlines. In *ICTAI '07 : Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence*, pages 231–238. IEEE Computer Society.

[Xu et al. 2008] Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). Satzilla : Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32 :565–606.

[Zilberstein 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3) :73–83.

[Zilberstein and Russell 1996] Zilberstein, S. and Russell, S. J. (1996). Optimal composition of real-time systems. *Artif. Intell.*, 82(1-2) :181–213.

Annexe A

A.1 Choix de T et γ dans la preuve de NP Complétude du *l-dRSSP* quand $k < m$

T a été choisi en vue de donner le coût de résolution des instances lorsque : l'on donne km ressources à A_{k+1} et l'on choisit un sous ensemble couvrant d'algorithmes (qui peut résoudre chaque instance en au plus $m\alpha$)dont chaque algorithme a une ressource. Le coût de résolution de I_{k+1} induit par l'affectation de km ressources à A_{k+1} est $\frac{k\gamma(k+1)m}{km} = \gamma(k+1)$. Le coût induit pour la résolution des n instances $I_1, \ldots, I_n$ avec un sous ensemble couvrant d'algorithmes ayant 1 ressource est $nm\alpha(k+1)$. On en déduit la valeur de T.

Pour choisir γ, nous considérons le coût T_1 de résolution des $n+1$ instances si l'on donne $km-1$ ressources à A_{k+1}. On peut facilement vérifier que $T_1 > \frac{\gamma(k+1)m}{km-1}$. Comme nous voulons que l'affectation de moins de km ressources ne puisse pas permettre d'atteindre une somme total de coût de T, nous posons $\frac{\gamma(k+1)m}{km-1} > \gamma(k+1) + nm(k+1)\alpha$ ce qui donne $\gamma > nm\alpha(km-1)$.

A.2 Choix de T et β dans la preuve de NP Complétude du *lr-dRSSP*

T est choisi ici de sorte à donner le coût de résolution lorsque l'on a un sous ensemble couvrant de x algorithmes (qui peut résoudre chaque instance $I_1, \ldots, I_n$ en au plus $m\alpha$) dont chaque algorithme possède deux ressources. Dans ce cas en effet, le coût de résolution des instances $I_1, \ldots, I_n$ est de $n\alpha\frac{m}{2}$ car ces dernières seront résolues par les algorithmes du sous ensemble couvrant. Parmi les k instances restantes $I_{n+1}, \ldots, I_{n+k}$, l'on aura x instances qui seront résolues par les algorithmes du sous ensemble couvrant et les $k-x$ instances restantes par les autres algorithmes. On en déduit que $T = nm\frac{\alpha}{2} + \beta(k - \frac{x}{2})$.

Pour choisir β, nous imposons que dans toute solution, il n'y ait pas plus de $k-x$ algorithmes ayant une ressource (pour garantir que l'on a bien x algorithmes ayant deux ressources). Si

$k-x+j$ ont deux ressources alors, parmi les instances $I_{n+1},\ldots,I_{n+k}$, $k-x+j$ sont résolues par un algorithme ayant une ressource et $x-j$ par un algorithme ayant plus d'une ressource. Ce coût est donc de la forme $\beta m(k-x+j)+t(x-j)$ où $t(x-j)$ est le coût des instances résolues avec plus d'une ressource. On peut facilement observer que la plus petite valeur de $t(x-j)$ est obtenue quand les $m-(k-x+j)=2x-j$ ressources restantes sont partagées équitablement entre algorithmes. Aussi on déduit que $t(x-j) \geq (x-j)m\beta\frac{x-j}{2x-j}$. Nous déduisons β en imposant que le coût de résolution des instances $I_{n+1},\ldots,I_{n+k}$ dépasse T quand on n'a pas x algorithmes ayant deux ressources. On pose pour cela l'inéquation $\beta m(k-x+j)+\frac{(x-j)^2 m\beta}{2x-j} > T$. Une solution existe dans cette inéquation si $\beta > \frac{nm\frac{\alpha}{2}(2(2x-j))}{jx}$.